JTC 1/SC 22/WG 14

document: N2522
version: CORE *202005*
date: *2020-05-10*

Programming languages — a common C/C++ core specification

Jens Gustedt – INRIA, France

Change history

v1: WG14 document N2494

v2: WG14 document N2522, this version, diffmarks from “cmin”

- clarifications: lambdas and `longjmp`, `inline` and scope, type `char8_t`, generic selection, `core::unsequenced` attribute
- three-way comparison (spaceship operator)
- “initializer” construct for captures of lambdas
- a tool for textual representation of all basic types and arrays, `totext`
- more attributes for allocated storage, `core::free`, `core::realloc`, `core::noleak`
- an attribute for tracking (or not) of initializations `core::writethrough`
- `core::concurrent` attribute
- `constexpr` based on `core::concurrent` and annotation of some library calls with that specifier
- `for`-loop and `if` with variable definitions
- harmonization of tag names between C and C++
- hammonization of the requirements for Unicode as an internal encoding

(no diff marks, here)

Abstract

The C and C++ programming languages have evolved from a common ancestor many years ago and have always been developed in keeping a close eye on each other. Both responsible committees, WG14 and WG21, have always sought both languages to be compatible as far as seemed possible; on a binary level for mutual linkage of software components, and on a source-header level for mutual access to the so linked components. Nevertheless, gratuitous incompatibilities have crept into them, and cross-language programming is nowadays quite difficult to achieve and almost impossible to teach.

On the positive side, in recent years the efforts to bridge the gap between the two language have been renewed and several fruitful initiatives have been undertaken to unify the approaches in several domains. These concern in particular atomic types and operations, sign representations of integers, the memory model(s), and the attribute feature.

*This specification is an attempt to strengthen these dynamics and to formulate a common language core that ideally would be integrated in both languages and would provide a solid base for the future development of both, **and**, that would be much simpler to use, to comprehend and to implement. It is oriented to maintain and extend some principal characteristics that are already present in the intersection:*

— *Strong static typing*

— *Type-genericity*

— *Efficiency*

— *Portability*

This common core adds features to both languages, and thus it has not yet a complete implementation. Nevertheless, first experiments show that it should not be very complicated to provide reference implementations within compiler frameworks that already have front-ends for both languages, and that thus already have most of the features in one way or another.

Acknowledgments

Discussions with the following people (and probably others that I forgot to mention) contributed to this proposal: Aaron Ballmann, Hubert Tong, JF Bastien, Kayvan Memarian, Lars Gullik Bjønnes, Martin Uecker, Niall Douglas, Peter Sewell.

(no diff marks, here)

Contents

Rationale

This document specifies the form and establishes the interpretation of programs expressed in a future core language specification common to the programming languages C and C++. Its purpose is to promote portability, reliability, maintainability, and efficient execution of programs written within that core on a variety of computing systems.

Clauses are included that detail the core itself and the contents of the C language execution library that is common to that core. Annexes summarize aspects of both of them, and enumerate factors that influence the portability of programs.

This document marks changes from version cmin to version core. They are indicated by ~~striking out~~ text that has been deleted and underlining text that has been added. Pages that contain changes are marked with cmin..core and are listed under core CHANGES in the index.

The starting point of this document was ISO/IEC 9989:2018. We apply a series of changes to the C programming language that are intended to ease the programming styles that are in synchronization with modern advances in information technology and software engineering, and that augment the common intersection with the programming language C++. As such, this core specification is currently neither conforming to the C nor the C++ standard, but the intent is that the C and C++ standards as well as this document are subsequently modified until they converge to a common core.

Changes that are integrated in this document come in several flavors:

— Changes that already have been integrated by WG14 into C for the upcoming specification of C2x. Beware that not all changes that WG14 has integrated are reproduced here. In particular, a lot of the changes to floating point types and libraries are currently too complex to be integrated in a core language specification.

— Features that have been present in C++, that are well established and that ease the portability of code between the two languages.

— Similarly, features that have been present in C for some time, that are commonly used and that ease portability with C++.

— Features that modernize the language specification appropriately to the improved environments that are nowadays commonly available.

— Disambiguations of problematic parts of both languages, in particular a consistent and comprehensive memory and aliasing model.

— Convergence of features that have gratuitously distinct syntax in C and C++, such as atomics and complex numbers.

— "Removal" of rarely used or underspecified parts of the languages that break compatibility between C and C++. Here, removal means "removal from this specification" and **not** removal from the corresponding language(s).

I C++ features

I.i Keywords

C has historically integrated some features that were invented for C++ but with a different syntax and sometimes even slightly different semantic. It did this by using a spelling for keywords starting with an underscore and a capital letter to avoid clashes with user space identifiers, and by only introducing the "normal" form (that would be a keyword in C++) via additional headers. This overcautious approach is in opposition to additions to the C library, where function names have been added without the same precaution.

Since these C++ keywords are present even in the C standard, user code targeting the C/C++ core may reasonably expect to be able to use these keywords without precaution. Therefore this core proposal moves forward to integrate them also as keywords to C and to deprecate the then useless header files that provide them.

The transition to that new setting should be as smooth as possible, so for the moment we keep the possibility that these keywords may be implemented as macros.

Another set of keywords that is introduced in this specification are eleven keywords that replace punctuators that in some historic settings had been difficult to represent. These have been keywords for C++ since the beginning and to accommodate C++ legacy code that might use them, we introduce them, here. This addition also makes the header `<iso646.h>` obsolete.

We also add new features implemented through keywords that had not yet been present in C, yet. Here also, to ease the transition it seems to be best to allow for them to be implemented as macros.

The keywords introduced by this change (6.4.1) are

alignas	**bool**	**not**	**true**
alignof	**compl**	**nullptr**	**xor_eq**
and_eq	**decltype**	**or_eq**	**xor**
and	**false**	**or**	
bitand	**generic_selection**	**static_assert**	
bitor	**not_eq**	**thread_local**	

Here the only semantic change that is necessary here is to impose that the type of the Boolean constants **false** and **true** is **bool** and not **int** as it had been in C. The new features are **nullptr** and **decltype**.

Overall, the C library headers that become obsolete are

`<iso646.h>`	`<stdalign.h>`	`<stdbool.h>`	`<stddef.h>`

Other C keywords follow a similar pattern, but the resolution of conflicting semantics is more complicated and will be handled below.

_Atomic	**_Generic**	**_Noreturn**
_Complex	**_Imaginary**	

I.ii Types and other fundamental language features

C and C++ differ in the presentation of certain semantic types that occur independently as language features, and this proposal attempts to provide a unified approach for them. Therefore the definition of these types is withdrawn from C library headers and definitions for these types are predefined. This allows to reconcile the fact that C has these types as **typedef** to basic types whereas for C++ some of them constitute proper types of their own.

We also add the type of the new **nullptr** constant to that list, 6.4.2.2. The types that become universally available (6.2.5.1) are:

Type	language feature
nullptr_t	the type of the **nullptr** constant
ptrdiff_t	the result of pointer difference
size_t	the result of **sizeof**, **alignof** and **offsetof** operations
wchar_t	the element type of **L** wide-string literals
char8_t	the element type of **u8** string literals
char16_t	the element type of **u** wide-string literals
char32_t	the element type of **U** wide-string literals

The encodings for the latter three are fixed for C++ to be UTF–8, UTF–16 and UTF–32, and so we remove the parts of the C standard that made such encodings implementation-defined. Also we unify the approach for internal representation of extended character sets such that implementations have to behave "as-if" the internal representation were Unicode. This ensures that the semantic of all characters in these extended character sets is unique and that mapping extended characters to Unicode and back is the identity.

In addition to these types, some other principal language features that in C were provided as macros have been elevated to be predefined macros. As a consequence the `<stddef.h>` header also becomes empty and is declared obsolete.

I.iii Compile time constants and well defined behavior

In contrast to C++, C is missing an important feature, namely the possibility to declare named constants of arbitrary type. The only possibilities that are currently offered are enumeration constants or macros.

Unfortunately, some of the paths that C++ has chosen to define such constants are not compatible with C. In particular, for C++ **`const`**-qualified objects (that are not **`volatile`**-qualified) and that have a known compile-time initializer are implicitly **`static`** and can stand in wherever a constant can. Using that construct in C by defining such a **`const`**-object in a header would introduce binary incompatibilities with existing code, and would thus not a path that C could easily chose to adapt. Also, that would instantiate such an object in every translation unit that includes the header, and it would results in multiple definitions if several such translation units are link into one executable.

The path chosen for this proposal is thus to use two features that are not yet present as such in C, namely *inline constants* and **`constexpr`**, see 6.6. The first are special cases of qualified inline objects, see below in IV.i, 6.7.5. They provide a possibility to define constants that have a unique address in the whole program, if such an address is needed.

`constexpr` are much more general feature, because this specifier can be applied to objects (similar to **`inline`** constants) but also to functions and lambdas. Thereby it allows to advance almost arbitrary computations to compile time **and** to ensure that such a computation only has defined behavior. The properties for that specifier are relatively involved when spelled out in full, although the most common cases are quite intuitive and can be verified automatically. We base the "implementation" of that feature heavily on the **`core::concurrent`** attribute, see below.

I.iv Empty default initializers

In contrast to C, C++ allows empty braces`{}` as default initializers for all value types. C only has `{ 0 }` to initialize the very first (maybe recursive) member to `0` and then all other members to their default. The later feature is commonly misunderstood and some compilers warn about this construct when it is used to initialize a nested aggregate or union type. Therefore we think that such empty initializers should be added to the C/C++ core. C implementations are invited to implement this as an extension to C for the time being (and some do that already).

Such a feature also comes handy for types that have no value but that should nevertheless be initialized. See opaque types (IV.iii) below.

I.v Specific named constants

This proposal adds the following specific named constants as language features: **`false`**, **`nullptr`** and **`true`**.

— In contrast to traditional C the Boolean constants have type **`bool`**.

— **`nullptr`** is meant to replace all uses of **`NULL`** and `0` for the purpose of specifying a universal null pointer constant. It is considered an error to use it in arithmetic and also to pass it as an argument of a function where there is no prototype.

I.vi Type inference and `decltype`

One big hurdle in C to program type-generic macros is the lack of tools that allow to infer the type of a variable from another one or from the type of an expression. We introduce two such tools, type inference (AKA **`auto`** feature) and **`decltype`**.

I.vi.i Type inference

Inferring type information from initializers, at least partially, is not completely new in C. In particular, incomplete array types can be completed by an initializer that is used to determine the size of the defined array.

Also, historically C had a default of "all **int**" for type declarations that were underspecified. In fact, syntactically the possibility not to have a *type specifier* was present in all versions of C, as long as the grammatical deduction was unambiguous; it was simply forbidden by a constraint to form such "typeless" declarations. This disambiguation of assignments and declarations, *e.g*, is possible whenever a storage class specifier is provided for a declaration. So a possible implementation of an inference of the type of a variable from its initializer is to simply allow to omit the *type specifier* and to set up rules how the type is inferred.

The rules to infer the type are also easy to set up: since we want to be compatible between C and C++ we have translated C++'s rules into C. It happens to be that these rules are very much the same as for type generic expressions **generic_selection**. So we abstracted these rules into a new feature *generic type*: the generic type is the result type that we obtain when we do array and function decay and drop all qualifiers from other types.

This idea of type inference even works for return types of functions. For functions that have a storage class specifier, *e.g* **auto**, the return type can be inferred from the arguments to the **return** statements, if there are any. The only constraint here is that the expressions in different **return** statements must not only be compatible, but have the same generic type.

The integration in the C language would then straight forward, wouldn't it be for the fact that C++ extended the lexical "role" of the **auto** to be a kind of "placeholder" for the unknown type. This is unnecessary for C, but doesn't hurt much either. So we basically allow **auto** to appear along with other storage class specifiers.

I.vi.ii `decltype` specifiers

Type inference only works with *values* and not *lvalues* and uses initializers for this. Therefore atomic-specifications get lost, and the type expression of interest must be such that it can be evaluated. Such constraints are sometimes too strong, for example if the type in question does not allow for evaluation or value initialization (*e.g* a **mtx_t**, see opaque types below) or if an automatic variable from an outer scope cannot be evaluated (see lambdas below).

Since almost a decade C++ has introduced the **decltype** feature for this. A **decltype** specifier is just a placeholder for a type, similar to a **typedef**. It reproduces the type "as-is" without dropping qualifiers and without decaying functions or arrays. With this feature not only qualifiers and atomics do not get dropped, but they can even be added.

Conceptually, integration into C is a bit more difficult than for type inference. This is because for historic reasons C++ here mixes several concepts in an unfortunate way: for some types of expressions **decltype** has a reference type for others it hasn't. The line of when it does this is not where one would expect it to be: most lvalues produce a reference type, but not all of them. In particular, direct identification of variables or functions (by identifier) or of structure or union members leads to direct types, without reference, but surrounding them with an expression that conserves their "lvalueness" adds a reference to the type of the **decltype** specification.

It is quite unusual for C to have the type of an expression depend on surrounding `()`, but unfortunately that ship has sailed. We try to mimic the behavior of C++ as close as possible such that we capture all cases where the resulting type has no reference. The goal is to have the intersection of the two languages as wide as possible.

I.vii Lambda expressions and function literals

Even in quite simple cases such as for a maximum operation, implementing type-generic macros often requires extensions to the C language. This is because it is difficult to define local objects within a macro that help to avoid multiple evaluation of macro arguments, and that don't create naming conflicts with existing identifiers. Also, C's **generic_selection** has several drawbacks for function-like interfaces that make it difficult to use beyond interfaces like `tgmath.h`, but see below for contexts where it actually is quite helpful:

- All possible type interpretation must be foreseen, new types can't be added easily.
- All case expressions must be valid for all cases, even for those for which they are not evaluated.
- Type generic macros with several parameters often need a large set of combinations of cases to be covered.

Another feature that is missing in current C, is the possibility to annotate and control specific block statements. There is no syntax to specify that a block should not be jumped into (for example into a loop body), to tighten the data flow that enters into such a block or to suspend strict type interpretation of objects for a controlled span of the execution.

Among the different extensions that implementations have provided to ease type-generic programming, C++'s lambdas seem to be the most promising. For their simpler variant (without references) they don't assume sophisticated heap-memory management, but can work with copies of lambda values representing "frozen" state on the stack.

We propose a relatively straight forward implementation of lambdas, 6.5.2.6. Here some specific subset of lambdas "function literals" play a special role. Function literals are the simplest type of lambdas that don't capture any automatic variables from the surrounding scopes. They basically behave as if they were unnamed **static inline** functions and are convertible into classical function pointers.

Other lambdas may capture automatic variables in surrounding scopes, but the access is restricted to copies of the variables in read-only mode. The type of such lambdas cannot be declared, there is no syntax for it, but it can only be inferred or specified through a **decltype** specifier. Thereby implementations are free to chose the implementation that suits them best, and reduce the overhead to a minimum. Lambda values themselves allow only two types of operations, copying via initialization or assignment, and function call. Conversion to function pointers is restricted to function literals.

Return types of lambdas in this proposal are always inferred. That is, there is no syntax to specify a return type for a lambda, and so the only way to specify one is the inferred type of all **return** expressions.

Since lambdas are expressions and not declarations, we can also allow type inference for the parameters. If such "generic lambdas" also have captures, they can only be used directly in function calls; the types of their parameters can then be inferred from their arguments. This allows to use them much as traditional `template` functions would be used in traditional C++.

If a generic lambda λ is also a function literal (has no captures) we can add another operation to the tools. Such a generic function literals λ can in fact be converted to any function pointer that presents a possible prototype for it. The proposal has a relatively simple example for a generic sort macro, `SORT`, that can be called with any type that has a `<` comparison defined for it.

Syntactically, lambdas from C++ are a bit weird, and used with their full potential they are getting even weirder. Together with VLA and attributes they introduce lexical ambiguity which is usually avoided for C:

```
const int val = 5;
const int * deprecated = &val;
double U[[deprecated]{ return deprecated; }[0]];  // U is a VLA, double[5]
double V[[deprecated]];                            // V is a deprecated double.
```

It is only at the second `]` or the `{` that the parser may distinguish if it is in the middle of parsing a lambda that would be the size of an array, or if this is an attribute.

This kind of ambiguity can be lifted by using more diverse technical character sets. See below for a proposal that allows to write the declaration of `V` with special double brackets:

```
double V ⟦ deprecated ⟧ ;
```

II C features

II.i Generic selection

C's dedicated feature to program type-generic interfaces has been *generic selection* with a relatively special syntax introduced by the `_Generic` keyword and `case`-like choices according to the generic type of the controlling expression. The important feature here is that such a generic selection is determined at compile time and the resulting type of the expression is the type of the chosen expression.

This choice of types is much more powerful than the one that can be done by generic lambdas:

— For lambdas, the result type basically depends on established type derivation mechanism. It is not easy to add such a mechanism that would not have been foreseen by predefined type conversion rules.

— The controlling expression of a generic selection is not evaluated, and neither are those that only occur in result expressions that are themselves not evaluated. Any mechanism that works with lambdas or function overloading will normally evaluate such arguments for their side effects.

A good example of the features that can be implemented with generic selection are type traits:

```
#define is_real_floating_type(X) \
      generic_selection((X),     \
            float: true,         \
            double: true,        \
            long double: true,   \
            default: false)
```

or value and type macros

```
#define unsigned_zero(X)              \
      generic_selection((X),          \
            long: 0UL,                \
            unsigned long: 0UL,       \
            long long: 0ULL,          \
            unsigned long long: 0ULL, \
            default: 0U)
#define unsigned_type(X) decltype(unsigned_zero(X))
```

Currently C++ seems not to have a tool that can easily emulate this, in the contrary implementing features that emulate multi-parameter generic selection is quite tedious. Usually the default case gives rise to a function or class template, and the individual cases then are specializations, but then additional boilerplate is needed to inhibit the evaluation of the controlling expression and more generally of those arguments that should not be evaluated, either.

So such a feature should be added to the C/C++ core. To increase the acceptability of the feature, we propose to rename the feature to `generic_selection` such that its purpose is more evident to the untrained reader. Also, having it as a newly named feature would enable us to enforce some more properties for the evaluation of the chosen result expression, if the discussion around this proposal reveals that this could be usefull.

II.ii Variable length arrays (VLA)

Traditionally, C and C++ differ in some of the aspects of array declarations, namely for arrays for which the bounds are not integer constant expressions (ICE). Generally (but see below) C allows them in block scope, whereas C++ has no such concept. C calls them *variable length arrays*, VLA, and pointers to such types are *variably modified* types, VM. These features and the difference between C and C++ has lead to endless debade, but it is commonly much misunderstood for its potential.

On one hand, VLA definitions in block scope can be dangerous, because they can lead to safety and security issues: they can smash the execution stack of functions, maybe inadvertently, or maybe even maliciously.

On the other hand, declarations of VLA (not necessarily definitions) are a convenient tool to enforce propagation of array sizes. In particular such an enforcement is possible from the caller of a function with array parameters into the function body, without changing function ABIs, without forcing transfer of dynamically allocated type descriptions, and without jeopardizing performance or safety.

C has VM types since C99, but made them optional with a feature macro `__STDC_NO_VLA__` in C11. This possibility not withstanding, there is no known implementation that would conform to C17 that defines that feature macro. C++ has no VM types. VM types, with the leeway for implementations to forbid definitions of VLA in block scope, are nevertheless proposed for this core specification, because they are fundamental for modern programming in C and because of the possibilities of array bound propagation, see Section III.iv.

Implementations may still opt-out from **defining** VLA by defining the feature macro `__CORE_NO_VLA__`, see 6.10.8.2.

II.iii Complex arithmetic

C++ is quite restrictive for the arithmetic operations that it defines for complex types, namely the other operand cannot not be a real type. Allowing such arithmetic is harmless because usually no information loss can originate from such an operation. Therefore this specification explicitly defines all four basic arithmetic operations that have both operands as arithmetic types. C++ implementations that want to target the common C/C++ core must either change their rules for arithmetic conversion or provide a series or overloaded `operator` functions that implement all possible operations for the for standard arithmetic operators.

III Modernization

III.i Mathematical functions

The `<math.h>` header has accumulated a lot of baggage over the years and introduces a lot of identifiers that are not protected by any naming convention. In the beginnings of C such an approach was adequate, because it was useful to have linker symbols for different variants of functions around.

Times have changed and the generic tools we propose here (inference, lambdas) go far beyond what had been possible, formerly. They make the need for such heavy intrusion in the users name space disappear.

In particular here we propose to replace most mathematical functions by type-generic macros, much as they are overloaded functions for C++. Basically this covers all functions that previously had been interfaced by the `<tgmath.h>` header. Compared to that header we introduce a big advantage: type-generic macros that are implemented with lambdas (or as-if implemented with lambdas) can be assigned to function pointers, such that applications can move function pointers around when they need them (e.g to compute derivatives). By this change

— most of the individual functions in `<math.h>` become obsolete,

— together with the changes on `<complex.h>` the whole `<tgmath.h>` header becomes obsolete, too.

There are some particular functions, where we go even a bit further, namely **`fabs`**, **`fmax`**, **`fmin`**, **`fdim`**, **`abs`** and **`div`**. These are all functions that present more generic language features than they are library features, and for which some historic choices have gone wrong.

For **`fabs`** and **`abs`**, there is first of all no real reason to distinguish floating point and integer interfaces. Mathematically it is clear what all these functions should do, and users can expect to have a single easy to use interface to address that feature. Second, **`abs`** had gone quite wrong for integer types: in some case there are cases where calls are undefined, simply because the historic choice for the return type was wrong. They probably date back in times where there were no unsigned types in

C or where unsigned types could just mask out the sign bit. So the choice then had forcibly was a return type that could not hold the absolute value for the minimum integer values in all cases.

The tide has turned, and today with the restrictions on sign representations that are in place now, there is a set of return types that can hold the mathematical values, so we should just chose this: we can simply force unsigned return values for all integer types.

Similar observations hold for maximum, minimum, and cut-off difference (`fdim`). With a proper choice of return types, all these functions can be specified without error conditions.

The `div` functions are even more peculiar. Currently each of them needs a proper return type and pollutes the name space with these mostly useless identifiers. We propose to change these into one single type-generic macro for which the return types then can be inferred. Most likely nobody ever is interested in the return of these functions for longer than some lines of code, so `auto` definitions of objects that capture the results should be fine.

III.ii Complex types

Although they are ABI compatible (have the same representation), complex types are handled quite differently in C and C++. In C there is the `_Complex` keyword that is used to specify complex types, in C++ there are templates `complex< F >` for all real types. Syntactically it would be difficult to reconcile these, so we don't even try.

Instead, we go the way of most modern programming languages by requiring them as mandatory builtin types. We introduce complex literals (with an additional `i` or `I` in the suffix) and as a consequence the complex types could simply be deduced by `decltype` specifications.

Predefined macros are added to deal with these constructs more comfortably: there are type macros (`real_type(T)` and `complex_type(T)`) and value macros (`real_value(x)` and `imaginary_value(x)`). We assume that such macro names (with `_type` or `_value`) will not produce to much conflicts in user space.

Some of the basic type-generic macros in `<math.h>` use complex arguments, without the need to include the `<complex.h>` header, namely `abs`, `conj`, `carg`, and `cproj`. (The later might still be subject to some name changes.) The functions `creal` and `cimag` are dropped bcause they are superseded by the macros above.

As a consequence of these changes for the complex types and the `<math.h>` the `<complex.h>` header can now be much simpler. It does not have to provide basic features for the types, and the interfaces are only amendments to the corresponding interfaces in `<math.h>`. There is a new feature test macro `__CORE_NO_COMPLEX__` that should be set if the functional interfaces are not provided.

III.iii Function attributes

The recent addition of the attribute feature to C makes it possible to add specific common attributes to both languages that may overcome the lack of precision for function and lambda interfaces that the languages traditionally provide. In particular the information about a function that the translator traditionally receives is limited to the parameters and to the return type, but completely ignores the rest of the program state. Modern optimizers are able to process much more information if functions and lambdas are annotated appropriately and produce executables that may perform orders of magnitude better.

The new attributes defined by this specification provide, 6.7.15.4, such optimization opportunities for functions and lambdas. Their main goal is to provide the translator with information about the access of functions and objects coming from surrounding scopes and such that it may deduce certified properties. This certification is ensured by forcing the attributes to be consistently present at all declarations, and to force the same type of attributes on other functions or lambdas that are called in the function body.

A first pair of attributes, `core::noleak` and `core::address_independent`, makes assertions about the behavior of functions with respect to the address space. The first guarantees that the function will not leak any allocation, that is, that every newly allocated storage instance will either be deallocated within the same function call, or a pointer to it will be returned as a `core::noalias` pointer. The second, forbids any exposure of storage instances or synthesis of pointers, and thus guarantees that the execution of the function is independent of any properties of the address space or of any particular address choices of any specific execution.

One set of attributes, `core::evaluates` and `core::modifies`, works with visible identifiers and establishes a strict framework of data flow from static or thread-local objects in and out of the function body. In addition, the `core::stateless` attribute guarantees that a function or lambda can not hold hidden state in form of a local static or thread-local variable. The second set, `core::state_invariant`, `core::state_conserving` and `core::state_transparent` go beyond this by controlling not only which identifiers are accessed directly, but also which objects are accessed through pointer indirections. Then, there are `core::idempotent`, `core::independent` and `core::unsequenced`, that are the most interesting attributes for optimization, but which can themselves not easily asserted through syntax and strong typing.

We also propose a more narrowly targeted attribute, `reentrant`, for signal handlers and functions that are used by them. Though this property can not be deduced automatically in all cases, it should be capable to check many candidate functions for signal handlers without user intervention.

The `core::concurrent` attribute describes a quite restricted set of functions or lambdas, too. Such functions and lambdas that may be robustly executed within different threads of execution without race conditions. Often they can even be performed once and for all at compile time if the arguments are constant expressions, and therefore this attribute also forms a main ingreedient for the formulation of the properties of the `constexpr` specifier for function definitions and lambda expressions.

The specification of the `core::evaluates` and `core::modifies` attributes use the names of the global objects that are accessed by the annotated function. Unfortunately, not all global state in the C library is identifiable by such a name. Therefore we extend the identifiers that are admissible to a set of placeholder names (such as `errno` or `stdout`) that we call the C library channels. Therefore the C library has been systematically combed for functions that make assumptions about a global state, and hopefully all have been annotated with the corresponding attributes.

Additionally, there are also aliasing attributes `core::noalias` and `core::alias`, that are also function attributes, but deal with much more, see Section V.iii for explicit aliasing handling, and the `core::reinterpret` attribute to handle type interpretation on function boundaries, see Sections III.iv and V.iv, below.

It is tedious to update large header files with these attributes. For cases were they a all the same for a whole set of functions we provide a `#pragma CORE FUNCTION_ATTRIBUTE` that can apply a pragma with arguments or switch it off if necessary.

III.iv Array size propagation

One of the worst traps that C and C++ have to offer, originate in the ambiguity between pointers and arrays, namely that pointers are supposed to point to an array of the base type, but where the size of that array is not known. This is particularly striking on the function call boundary, where arrays are rewritten to pointers on both sides:

— On the definition side array and pointer parameters are "considered the same" in a very weird way, namely most information that may even be present in the array specification is pretended to be lost the moment we enter the function.

— On the calling side, arrays "decay" to pointers, and any information that might even present in the interface, such as array sizes are not not enforced.

All of this is not only dangerous, it is also completely useless. Nowadays in many situations there is not even a performance gain produced by these "features". So we think that it is time to tighten the rules such than array sizes can be propagated and checked without otherwise harming performance or even productivity.

The idea for this is simple: enforce that function declarations are consistent, in particular that specified array size expressions are the "same". Here the same is modeled by something coined *token equivalence*, that is were declarations are equivalent as token sequences, with the possibility to (re-)name function parameters and to adjust white-space and digraphs. But for example, array-to-pointer rewrite would not be allowed in the declaration of a function were the definition would be written with array notation.

Token equivalence warrants that both, the caller and the definition, see the same expressions for array bounds. Thus the caller can check such conditions at compile time (or maybe at run-time) and the definition may safely assume that the condition has been verified before any call.

Several mechanisms are put in place to ease array size propagation. First there is a function attribute `core::reinterpret` that (among other things) enforces that all declarations (including definitions) are token equivalent. Then, there are function return type annotations, such as `core::noalias(size)` for `malloc` or `realloc` that provide information about the size of the array their returned pointer refers. Third, the consequent use of VM types (see above) for array parameters, enforces that the translator must have a notion of a dynamic size that is associated to a pointer, and VM types can be used to propagate the information from assignment to assignment.

All of this is certainly not yet complete, and other tools will have to be added later that, on one hand, will ease such an analysis, and on the other will equip the programmer with tools to annotate declarations with size (or more general, pointer and aliasing) information.

III.v Qualifier fidelity

The C library has a lot of interfaces that can be used for write-privilege escalation: they accept pointers to `const`-qualified objects and return a derived pointer that drops the qualifier. At the time these were introduced, this was probably a good compromise for the usability of these interfaces; a pointer to a non-qualified object can be passed into such a function without explicit conversion, and then the return value still has the same qualification as the original. But, this technique has the disadvantage that pointers to objects that are genuinely `const`-qualified, are then exposed with a pointer that has the qualification dropped.

With type-generic interfaces all of this can be easily avoided, if the return type of such functions is inferred from the argument.

Qualifier fidelity also has the advantage that generally arguments to such functions don't have to be converted at all. That is, for the (rare) case that objects are genuinely `volatile`-qualified, the semantics for such objects are respected. This can be particularly important for security sensible data, where applications must be guaranteed that copy or erasure operations on byte arrays are effectively performed through the whole memory hierarchy. In particular, the `memset` type-generic macro is guaranteed to overwrite a byte array that is passed in which has a `volatile` qualification.

III.vi Three-way comparison

Many other languages already have three-way comparison by means of an operator that is usually denoted as `<=>`. It provides an interesting abstraction for search and sorting interfaces, because here usually one wants to know three possible outcomes, if a value is less, equal (or equivalent) or greater than another value.

C++ has introduced such an operator recently, and it provides interesting features also for other aspects:

— It can be made well-defined. That is, the only errors that can happen are constraint violations that can be detected at compile time. No undefined behavior may result.

— They are composable, such that they can be extended to any aggregate type. Basically, for aggregate types they define lexicographic ordering.

For C this means that we can add comparisons that have the following properties.

— It sorts all valid pointers and null pointers, regardless if they point to the same array or not. Null pointers are here sorted as being smaller than any valid pointer, which relates with the fact that most platforms have null pointers as all bit zero representations nowadays. This allows, *e.g.*, to write checks for the intersection of arrays (provided by pointers), that would otherwise encounter UB in some cases.

— It provides structured equality tests for all types, not only basic types and pointers.

— It allows to provide simple interfaces for searching and sorting utilities that are type-safe.

III.vii Textual representations and output

C and C++ have quite diverging tools for textual output and for textual representation of data in general; C's work horse for human targeted IO is `printf` (and similar), whereas C++ mainly works with the shift operator `<<`. None is really suited to replace the other.

C++'s operator approach is not easily transposed to C, because C does not have operator overloading. Also, the possibilities to modify the textual representation that are provided via "manipulators" usually switch modes for a whole output operation, where the C tool allows a more fine grained handling of individual output items.

`printf` is clumsy and unsafe: it needs that the programmer manually maintains format strings. Type mismatch between format strings are a common programming mistake, and the possibility to have such format strings passed into `printf` dynamically opens a big security hole for stack attacks and similar.

Generally speaking, a basic interface for string handling and user IO should only have defined behavior and all programming errors should result in a constraint violation. This specification tries thus to propose one new tool, the `totext` type-generic macro, see 7.22.2.1, that has these properties. For its functionality, it is mainly based on the `snprintf` function, but without a requirement to maintain a consistent format information, and by avoiding to have to specify a compile time property as a string. It has several modes of operation:

— A simple mode for all basic types allows to generically store textual representations for numbers, `bool`, pointers and strings. It deduces the "format" for the operation from the type information.

— A string mode can be chosen to convert all three types of wide-strings to a textual multi-byte representation.

— A set of (integer) flags allow to adapt the desired output, for example to adjust the precision or to chose between different number bases or output formats. For example `bool` values may be represented as "0" and "1" or as `false` and `true`; a character pointer can be interpreted as pointing to a string or the address that it represents can be printed.

— An array mode allows to print entire arrays with a consistent set of flags and to separate the individual elements by a user provident `glue`.

The features are chosen carefully such that most of the operations can be used within `constexpr` function or lambdas. Exception from this rule are wide-string conversions because they need `locale` information and textual representations of pointer values because they expose storage instances.

Then, the existing interfaces `strlen`, `strdup`, `strndup`, `fputs` and `puts` are extended similarly. For example, `puts` prints the textual representation of its first argument followed by a newline to `stdout`. If that first argument is a character pointer, the behavior is the same as it has always been, but if it is a `double` the representation of the value is printed, if it is wide-string of some sort, the multi-byte encoding of that wide-string is printed etc.

In all, these new interfaces provide simple means for textual representation and output that have (mostly) defined behavior.

IV Disambiguation

In many places, C and C++ gratuitously differ in an annoying way, and unfortunately we will not be able to resolve these differences easily; too many code builds on such properties in one or the other language. For the features treated in the next sections we identified a need for action(s), because they are sufficiently central and important, such that there should be provided a way forward, now and today.

Many other features, are not yet handled, either because we did not find them important enough, or simply because there was no idea popping up on how to solve the problems. For these we added a lot of Notes and footnotes that attempt to expose the problem and provide recommendations how programmers that target the common C/C++ core should attempt to circumvent problems, and how implementations could make life for people easier. This treatment of the standards text is yet incomplete, and others will hopefully be added to this document over time.

IV.i Inline functions and objects

C and C++ differ slightly in their handling of inline functions. Whereas C enforces the use of an external definition in certain situations, in particular if the address of an inline function is used other than in a function call, C++ always guarantees that an external definition (called an instantiation) is emitted if there is need for it. This choice for C is deliberate, because traditionally C is often used in contexts that have severe constraints on the memory size for the program image. So a systematic generation of unused function definitions in all translation units is avoided.

This specification, 6.7.5, follows C++ (and extends C) by requiring that the effective semantics of inline and external definitions have to agree. It follows C (and extends C++) by requiring that no non-**`const`** qualified objects with internal linkage may be accessed by inline functions.

C currently has no inline objects, so this specification imposes an extension of the C language. The definitions presented here not only serve the purpose of programming invariantly in C and C++, but also to provide a tool to specify compile time constants of any object type.

For both, functions and objects, the choice has been made to follow mostly the C model for instantiation, that is, to require that an external definition must be presented explicitly for functions or objects that use the address or that form a modifiable lvalue. So this part of the specification extends the C++ language by imposing more constraints on well-formed programs. The specification of **`const`**-qualified objects allows to avoid the need for instantiations, if the address of the object is never used.

IV.ii Lexing of punctuators

C and C++ have different lexing rules that are not always compatible. Whereas C indifferently applies the “maximal crunch” rule, C++ partially implements semantical disambiguation, in particular for >> tokens. The problems behind the possible lexical ambiguities had been introduced at times where only a limited number of punctuation characters had been commonly available on computing devices. (And even then there were nasty problems with the heterogenity of platforms.)

In a world of Unicode such problems just disappear in thin air, and it would be preposterous to impose lexical acrobatics to future generations of C or C++ programmers, just because the standards had not been clear at the beginning. Therefore we propose to change the definitions of all punctuators that have reasonable definitions as Unicode points to such code points. This disambiguates the following constructs

prefix *	binary ×
prefix &	binary ∧
subscript or array size opening and lambda expression [[	attribute opening ⟦
nested array termination]]	attribute closing ⟧
nested template termination >>	right shift ⊠▷

and generally leads to code that is easier to read and to comprehend.

For backwards compatibility we propose to keep the old definitions such as << or <= as digraphs.

IV.iii Opaque types and `void` return from functions

A certain set of types has quite different treatment between C and C++, namely types that have no copyable value but only represent internal state. For C, these are **`fenv_t`**, **`fexcept_t`**, **`FILE`**, **`jmp_buf`**, **`va_list`** and types in the thread and atomic extensions that have specific macros or functions for initialization and that a priori cannot be copied. C has refused to provide sound semantics and is silent about how to treat them for example if they are copied byte-by-byte.

In C++, such types typically have default initializers that are called automatically without explicit mention in the code. Therefore we opted for the possibility of implicit and explicit initialization of these types. The concept invented for this is "opaque types", 6.2.5, that allows to capture types that have no value but an internal state that cannot be copied.

We also extend this concept of opaque types to **void** (with a size of 1) such that we can allow the definition of untyped byte arrays. On one hand, this permits to have statically allocated memory arenas that can be used with changing effective type, much as allocated memory, and on the other hand to annotate pointer arguments with no type (classicaly expressed as **void**∗ parameters) with a size (by using a fixed or variable-length array parameter **void**[`size`]).

Using that, it is also easy to extend the **return** statement, such that it may have an expression even if the return type of the function is **void**. This makes programming of type-generic macros and lambdas much easier, since it avoids a case analysis concerning return types.

IV.iv Atomics

C and C++ have no reconcilable syntax for specifying an atomic derivation: C has a keyword **_Atomic** that is applied as a specifier (similar to here) and as a qualifier, C++ has a class template `atomic`<*type-name*>. Since it even has ambiguities, sticking to the C syntax was not an option. The specification as given here has straight forward implementations in the old syntax for both languages: the type specifier **atomic_type**(`T`) can easily be set to **_Atomic**(`T`) for C and to `atomic`< `T` > for C++.

The specification of the atomics extension in the C standard has been surprisingly loose, ambiguous and incomplete. In order to become suitable for coding in the C/C++ core, a lot of cleanup work had to be integrated to the specification. The main properties of this extension are

- Clarification which operations are synchronization operations.
- Type-generic macros are added that cover all operations that previously only had been provided by operators such as multiplication or bitshift.
- Type-generic macros (operation and then fetch) have been added that provide exactly the same operations as the operators, and generalizes them to other **memory_order**s than **memory_order_seq_cst**.
- The specification of the type-generic macros has been extended such that they now behave like lambdas and may be converted to function pointers.

IV.v Bit-fields and fixed-width types

Both, C and C++, have the constructs of bit-fields that are conceptual objects on a scale below a storage unit. Unfortunately both disagree on their interpretation in terms of types and possible bounds to the number of bits. We provide a framework that is meant to cover the intersection of these features for the two languages. Therefore we use the concept of integers of a given width M, **intwidth**(M) and **uintwidth**(M). For these types we define simple rules how they are represented (basically with a size that corresponds to the best fitting types **int***N***_t**), and how they convert when used in expressions.

Second, we use the packing rules that are provided by the **core**::**noalias** and **core**::**alias**, see below, to describe how a bit-field "`name`:M" translates into a fixed-width integer of type **intwidth**(M) or **uintwidth**(M) with the proper attributes.

Otherwise, we restrict the admissible types for bit-fields to **bool**, **signed** or **unsigned**, because in particular the specification of **int** can be different between the two languages, and also because the only important information for integer types (that are not **bool**) are their signedness and their width. The only implementation-defined parameter for bit-fields is then the maximal admissible width for which we introduce the new feature test **INT_BITFIELD_MAX**.

IV.vi Identifiers

For several points, C and C++ have subtle differences in the way they handle identifiers, namely the scope **for**-loop variables or function parameters, tag names and :: -chained identifiers.

IV.vi.i Variables in for-loops

Here the difference is just an annoyance and only a historic artifact without much reason of existence. The problem is that for C, the block of a **for**-loop constitutes a new scope of its own, such that there are effectively two scopes, the one of the whole **for**-statement and the one of the loop body. For a large majority of code this makes not much of a difference; such a difference only manifests if the scope of the loop body declares a variable with the same name as the loop iterator declaration.

WG14 relatively recently had a proposal to fix this incompatibility to C++ but decided not to modify the C rules for backwards compatibility. But in the context of a common C/C++ core it makes no sense to stick to such a rule, because code that has a shadowing variable can never be ported as such to C++.

The choice for this specification was then to recommend a diagnostic for such situations or to implement the stronger C++ rules. We went for the latter because the diagnostic of such a situation is always possible, because for C a constraint violation has no enforced impact other than to require a diagnostic, anyhow, and because it seemed desirable that an implementation may refuse to produce an executable in such cases.

IV.vi.ii Visibility of parameters

C and C++ have different rules for the visibilty of function parameters: for C a parameter is visible starting at the end of its declaration, whereas for C++ it is only visible starting in the function body, if the declaration also happens to be a definition. This specification opted for the C variant, because this rule implies that one parameter can be used for the declaration of the type of another. That possibility is important wherever there is a need to ensure consistency between types or array lengths.

In C a typical usage of that feature is array bound propagation, as in

```
double dotprot(size_t n, double A[n], double B[n]);
```

but with this specification of the common C/C++ core the use goes further, for example by using a parameter that is already known within a **decltype** declaration of another parameter, or for using a parameter name in an `core::alias` or `core::noalias` attribute:

```
double dotprot(size_t n, double A[n], decltype(A) B);
void* malloc(size_t size) [[core::noalias(size)]];
C* memcpy(C *[[core::noalias]] s1, D *[[core::noalias]] s2, size_t n) [[core::alias(s1)]];
```

IV.vi.iii Use of tag names

C and C++ differ in the use of tag names (the identifiers in **struct**, **union** and **enum** declarators). C++ allows the tag name to stand in for the type where that is possible without ambiguity, whereas C clearly distinguishes “name spaces” (not to be mixed up with “namespaces” in C++) and does not allow such a use. It is not easy to adopt that policies for C, because there are examples in the wild where this feature is used, notably for the `stat` type and function in POSIX.

On the other hand, C and C++ have different policies concerning the reuse of tag names as identifiers. Whereas C allows an unrestricted use, C++ does not allow it for **typedef** (or `using`), unless the so specified type is effectively the type that has the tag.

As a compromise between those two sets of requirements we have adopted two measures:

1. Disallow the use of a visible tag for a **typedef** other than the tagged type.
2. Recommend the usage of trivial **typedef** that impose the introduction of a tag name as an alias to the same type.

IV.vi.iv Chained identifiers

C++ has a notation to access a member of a structure (class) or union, but without refering to an object, that is completely absent from C. It works with identifiers that are chained with a :: token. Translated into C an access as in the following

```
typedef struct A A;
typedef struct B B;
struct A { double a; };
struct B { A ba; };
...
sizeof(B::ba::a)
...
```

would be equivalent to

```
sizeof(((B){ }).ba.a)
```

that is, to create a compound literal of the requested type (the first element in the identifier chain, `B`) and then iteratively accessing the members of that compound literal (`ba` and `a`) with the `.` operator. C has no structure or union members that would be allowed in evaluations without having a concrete instance of such a type, and the use of such a construct would be restricted to contexts that are not evaluated, that is **`sizeof`**, **`alignof`**, and the controlling expression in a generic selection (plus **`decltype`** with this specification). Therefore, this feature seemed to be of minor importance for the common C/C++ core and was not added.

The usage of that feature is not conforming to the syntax of C and is therefore a constraint violation. All implementations that target the common C/C++ core should diagnose this use of the :: token.

V Memory model

Both, C and C++, historically have had difficulties in describing consistent and comprehensive memory models. Recently some effort has been made to accommodate these different models and to bring them in alignment among each other (C and C++), and amoung expectations of users and implementers. Therefore we apply modifications that try to simplify the existing C model and to disambiguate it. It has already found acceptance by part of the C and C++ committees, so there is hope that both languages converge to something that is similar as described by this document.

In particular, WG14 has recently agreed that a technical specification (TS) should be created for that model. This specification here aims to be synchronized with that TS. To say whether or not C or C++ already implement this model is moot, as the texts are ambiguous and there will be as many opinions about this as there are C and C++ experts.

One of the strengths of C is its efficient handling of aliasing, respectively of its capacity to deduce non-aliasing between given pointed-to objects, and to optimize code as a consequence. This property of the language is due to the combination of the following;

— Type based aliasing: besides some exceptions, pointers with different target types cannot alias.

— Provenance based aliasing: two pointers that come from different object definitions or calls to allocation functions cannot alias.

— Lack of references: most address-of operations are done explicitly.

— The **`restrict`** qualification of pointers allows to explicitly state the absence of aliasing between given pointers.

— The **`register`** storage class can be used to inhibit the taking of addresses. (This feature not used very often, though.)

These features have a lot of drawbacks, though. First of all, type-based aliasing (the effective type rule) is poorly specified and has many ambiguities at its margins. Second, provenance based aliasing analysis is not even properly spelled out at all, but buried in some obscure and inconsistent "answer to a defect report", that still as of today can trigger passionate but fruitless discussions about the turning of words and the world in general. Then there is the **restrict** qualification, that is only a specification for a function definition and not contractually binding for the interface. A user of a function can only successfully use the aliasing properties if it inspects the function body, the interface isn't enough.

V.i Storage instance

There is a lack of terminology to describe the entity that is reserved and released by either an allocation (**malloc/free**) or by the definition of a variable or compound literal. We introduce the new term *storage instance* to distinguish it clearly from the term *object*. We also use the opportunity introduce and clarify terminology as for the *start address*, *end address* of storage instances and similar concepts.

V.ii Provenance-based aliasing analysis

There has been reached wide consensus in the parts of the C and C++ committees that deal with these questions that an important component of the memory model should be provenance-based aliasing analysis. The idea is that two pointers that have different "origins" can never point to the same entity, and thus they always can be assumed not to alias.

The variant implemented here sticks to the granularity of storage instances, called *provenance*. It suggests that pointer arithmetic should never cross the boundaries of storage instances, and thus pointers that originate in different storage instances should normally not run into each other. As long as the application code does not play dirty tricks, see below, usual pointer arithmetic should always warrant this, and so under most circumstances a compiler should be able to assume the provenance of pointers to objects "as it sees them".

All of this should even hold for pointers that are "off by 1", that is, where the address is just after a storage instance. These appear relatively often in stop criteria for array traversals, but as long as they are used consistently and only compared with pointers with the same provenance, there should not be much of a problem.

There are several constructs that are identified as "dirty tricks" called out by this proposal as *exposing* a storage instance. These are all constructs that interpret pointers differently or leak information of their internal representation: pointer to integer casts, accesses to individual bytes of pointer representations, IO of pointer representations. Once the information about the address of a storage instance has been exposed, we cannot be sure that these addresses do not creep in incidentally or accidentally. So objects that live in such exposed storage instances need special care and only a much more restricted aliasing analysis can be performed with them.

V.iii Explicit aliasing deduction and storage allocation

The keywords **register** and **restrict** are absent from modern C++ and it seemed necessary to be able to propose new mechanisms, that can be introduced to both languages and that maintain or even extend the capacities for aliasing analysis. Additionally, a mechanism is needed to describe allocations. C has **malloc** and similar tools, C++ has `new` and `delete`, and these different tools are difficult to reconcile.

The tool chosen for these extensions are *attributes*. These were not much explored in the C++ standard itself, and have only recently be added to C. Attributes allow to add properties to interfaces, without necessarily extending the language. One set of attributes that help for the aliasing analysis have already been introduced above. They make the changes of the execution state made by a given function predictable and thus allow to draw certain conclusions about mutual aliasing (or not) of pointers. This set is extended by four additional attributes, **core**::**noalias** **core**::**alias**, **core**::**free** and **core**::**realloc** 6.7.15.3.

Depending to which construct it is applied, the **core**::**noalias** combines properties of C's **restrict** and **register**, and gcc's **malloc** attribute, and extends them. When applied to an identifier it is similar to restrict and forbid to take the address of that identifier, so this is similar to **register**. The application of that feature is more general, though, because it applies not only to block or function scope, but can also be applied for globals. There it is interesting to ensure that an object or function pointer can never escape from a translation unit and can thus be completely integrated in place. The second usage of `core::noalias` is drawn from `restrict` and its simplest usage is equivalent to that. More sophisticated usages allow to add a size argument to the attribute, such that the translator can infer overlap properties of arrays. Then, `core::noalias` also allows to to annotate a pointer-return from functions (such as `malloc`) as providing a *freshly* allocated storage instance. The latter notifies the translator that the result pointer will not alias with anything known so far.

A complementary attribute `core::alias` has the inverse role, it can provide with aliasing information, namely that the so annotated pointer aliases another one, in situations where such an information is not deducible by the translator. This concerns in particular return values from functions, that can be annotated for example with the names of the function parameters that they return.

The other two attributes, `core::free` and `core::realloc`, complement the allocation management aspects. Applied to a pointer parameter they indicate that the function behaves equivalent to `free` or `realloc`, respectively. Thereby, these attributes allow to completely specify interfaces that allocate, deallocate or reallocate storage instances, and enable the translator to actively track allocations, and use that information for example to ensure the `core::noleak` attribute as introduced above.

Both attributes `core::noalias` and `core::alias` are also used to specify packing rules for union or structure members. A member that has a `core::noalias` attribute cannot have its address taken, and therefore alignment constraints can be relaxed. By that, padding between members can be reduced and the effect is similar to the `packed` pragma that many implementations provide. The `core::alias` can then be used to describe members that potentially share the same storage unit, and that, if an address could be obtained, would alias each other.

V.iv Type-based aliasing analysis

Type-based aliasing analysis in C is a mess. It is guided by the "effective type rule" that, on the surface, promotes a simple idea: if types are effectively enforced, two pointers to different types can never alias each other. Unfortunately, the premise here is wrong, types are not enforced effectively, and the language has several loop holes to "legally" mess with the type system.

The implementations of this in the C standard and also in the field has failed dramatically. There are no two implementations out there that seem to interpret the rules consistently, and probably there isn't even one, that interprets them consistently within itself. Endless debates have not been able to solve the underlying issues, and also C's Memory Model Study Group has not yet been able to complete even a full analysis of the problems.

The main issues are

1. Types of allocated regions (via `malloc` et al.) are not fixed, and there is not even a concrete point in time when such an object acquires a types, or when such a type changes.
2. On the other hand, at least temporary reinterpretation of objects with some sort of typed-view is common practice and much needed. Currently used features are type punning through **`union`**, pointer type casts, passing a two-dimensional array as a one-dimensional one to a function, direct manipulations of bytes of representations and probably many more.
3. Types can be nested, so a particular region of memory can be part of a nested hierarchy of objects. There is no consensus so far how this type and object hierarchy can be visited, and which implication the implicit knowledge about one object being part of a bigger one can have on aliasing analysis.

Evidently, we cannot yet have a complete answer to all of these problems, but the `core::reinterpret` attribute gives a partial answer to 1 and 2. It forceably places the boundary of type interpretation at the level of a function or lambda call. The idea is that the translator of the definition always has the description of a parameters in the prototype, so it may just *assume* that these are the types of the underlying objects. The consistency of the `core::reinterpret` attribute is enforced in the interface (via token equivalence) such that there can be no denial: the caller knows what types the function expects and the definition clearly indicates what types it expects to find.

Because all of this happens at the call interface, there is no need, even conceptually, to trace a possible previous "effective type" of an object. The only properties that have to be ensured is that the data that the function finds has valid values for their parameters (viewed in the type they expect) and that any manipulation of the pointed to objects guarantees that the caller still sees valid values for the objects they happen to know.

The information about an object pointed-to by an argument/parameter pair that caller and function share is not only the type, but also the concrete representation of that type on a particular platform, and because of the enforced token-equivalence, the pointed-to size. So effectively the rules of matching argument/parameter types can be relaxed to a notion that we call "*equally represented*" types. This notion allows for example to pass a complex vector into a function that handles floating points, or temporarily see a table of `uint32_t` as `uint16_t` or vice-versa.

The introduction of lambdas to the common core makes this attribute much more powerful than it might look at a first glance. By reformulating a block of code as a lambda, the programmer can clearly indicate the input/output into such a block and the `core::reinterpret` attribute may then force a certain type interpretation that is well contained within the body of the lambda.

V.v Data-flow analysis

Hardware capacity and data-flow analysis of modern compilers has very much improved over the beginnings of C and C++. At that time, not providing default initialization for automatic variables of basic types was probably unavoidable: the lack of hardware registers and techniques for dead-store elimination called for performance improvements, and these were only possible by putting the burden of the identification of the first effective store operation on the programmer. This strategy of avoiding initialization has caused a lot of serious bugs over the years, and recent studies have shown that a lot of attact vectors may simply vanish, when initialization of all defined objects is enforced.

Things have much changed since these beginnings and other experiments show that situations were a translator may not efficiently identify the first effective store are rare, and quite particular. Basically there are three situations:

1. Initialization of an object is delegated to an external function, either by passing the address of the object (mostly C) or a reference to it (only C++) as an argument.

2. Initialization is hidden in control flow for which the translator is not able to prove validity, either because the control flow itself depends on execution state, or because only implicit conditions (but known to the programmer) guarantee a valid initialization in all cases.

3. A function returns a pointer or reference to an object that had been allocated by one of the storage allocation functions (`malloc` and friends) and the contents of that object has not been initialized.

All three situations have in common, that the translator has to deal with an lvalue which it can't convert, but on which, before any other use, it must first perform a *writethrough* operation, that is a store operation that stores without reading. For this specification we propose to capture this by the `core::writethrough` attribute, which is a annotation of objects, pointer parameters and return values, that allows programmers to emphasize that the lvalue in question is to be considered uninitialized. With that knowledge, translators may then suppress initial values (if any) or may ensure that objects that are passed into functions are effectively initialized in all cases.

Our hope is that such an attribute may help to switch from a model of "no default initialization for automatic variables" to "default initialization unless explicitly excluded", in a near future for both languages.

V.vi Const objects of static storage duration

Differences for the representation of `const`-qualified objects with static storage duration between C an C++ are subtle. In particular, the differences for such objects (such as `__func__`) that are defined in `inline` functions are not apparent at a first look. The differences are mainly historical and seem not have a value as such. The present specification tries to alleviate the coding with these differences and points out some of them as notes and by recommending certain diagnostics for implementations.

VI Removal

Some features are so diverging between the two languages, that a huge effort or even sacrifice would have to be made by one of them to able to compromise. We don't think that expecting such a convergence would be realistic, and we try thus to "remove" them from this specification. Here, removal really means "removal from this specification" and **not** removal from the corresponding language(s). Each of the languages should be able to handle their set of suplemental features all by themselves.

VI.i Lvalue expressions and reference types

C and C++ have very different strategies concerning the value category of expressions and C++ has even a construct that allows to create aliases of lvalues, namely references. C's tradition is to be much more restrictive with aliases (and aliasing) and therefore to drop the object information in expressions and types as soon as possible. This has allowed the C object model to remain relatively simple and to focus on computations in the abstract machine, instead of dragging representation and aliasing information through optimization phases. Modern C compilers are coping with some of the minor problems of this approach (such as copy elision) quite efficiently, and the hope is that the tools that are presented here (such as **`constexpr`** and lambdas) will help them even to improve on these possibilities.

Generally, for C++ expressions are *lvalues* whenever that is possible. In contrast to that, in C most operators undergo *lvalue conversion* (see 6.3.2.1) *before* they enter an expression and the information about the object(s) that entered into an expression is discarded. By that, a lot of expressions that are valid for C++ are not valid for C. *E.g* in C++ the prefix increment operator `++` can be applied multiple times in the same expression (`++ ++a`) or the ternary operator can be used on the left side of an assignment `(isit ? a : b)= 76;`. Both are invalid for C.

For C, lvalues only enter into expressions that are supposed to modify an object (such as assignment operators, increment and decrement), that compute its address (address-of operator), that access members (`.member` operator), or that query type properties such as size or alignment. The result of an expression is only an lvalue for the dereference operator `*` and for member access (`[]`, `.` and →).

Programming for the C/C++ core implies not to use such constructs and we volutarily keep the possibility of returning lvalues out of this core specification. So this specification "removes" the lvalue feature from the following operators, but note that this only concerns these operators when they are applied to basic types.

— prefix increment and decrement

— conditional operator

— assigment operators

— comma operator

Generally, the use of such constructs as modifiable lvalues should be rare and is widely considered to be poor programming style. All such uses have always been constraint violations in C, so they should be easy to diagnose, and this is what is expected from implementations that want to target the common C/C++ core.

VI.ii Conversions from complex types

C allows conversions from complex types to real types by simply dropping the imaginary part. This feature is not compatible with the more restrictive approach of C++ that basically requires all conversions that may result in information loss to be explicit. Therefore for the common C / C++ core the only defined coversion from a complex type to a real type is if the real type is **`bool`**, For arithmetic conversions that involve complex numbers see II.iii.

VI.iii Imaginary types and Annex G

C has reserved the `_Imaginary` keyword for optinal imaginary types and provides Annex G for a description of these. This has not found widespread support in the C community and has never been adapted to C++. Therefore these interfaces are not part of the C/C++ core.

VI.iv Bounds-checking interfaces (Annex K)

For C, there is a large and complicated annex (Annex K) that describes a set of extensions that are only scarcely implemented on real platforms and have a lot of issues. It has not found consensus in the C community and has never been adapted to C++. Therefore these interfaces are not part of the C/C++ core.

VII Further directions

Some work is still missing such that this proposal would be consistent in itself, and to integrate and to mutually adapt it and the two standards to which it relates. Additionally, we foresee to address the following features and questions:

— Introduce enumeration types with specified base type?

— How to handle `_Noreturn`?

— Do we want new syntax for number tokens, such as thousands separators or base 2?

— Do we keep qualifiers inside the `[]` array bounds for array parameters?

— Do we keep `static` in array declarations?

— Shall we treat padding as `void` arrays?

(no diff marks, here) *Foreword to be provided by the committee responsible for publishing.*

Introduction

1 With the introduction of new devices and extended character sets, new features could be added to this document. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, could conflict with future additions.

2 Certain features are *obsolescent*, which means that they could be considered for withdrawal in future revisions of this document. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.32]) is discouraged.

3 This document is divided into four major subdivisions:

— preliminary elements (Clauses 1–4);

— the characteristics of environments that translate and execute programs (Clause 5);

— the language syntax, constraints, and semantics (Clause 6);

— the library facilities (Clause 7).

4 Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this document. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes define optional features, provide additional information and summarize the information contained in this document. A bibliography lists documents that were referred to during the preparation of this document.

5 The language clause (Clause 6) is derived from "The C Reference Manual".

6 The library clause (Clause 7) is based on the *1984 /usr/group Standard*.

(no diff marks, here)

JTC 1/SC 22/WG 14

document: N2522
version: CORE *202005*
date: *2020-05-10*

1. Scope

1 This document specifies the form and establishes the interpretation of programs written in the ~~C programming language~~core of the C or C++ programming languages.[3] It specifies

— the representation of ~~C~~ programs;

— the syntax and constraints of the ~~C language~~core of the two languages;

— the semantic rules for interpreting ~~C~~ such programs;

— the representation of input data to be processed by C and C++ programs;

— the representation of output data produced by C and C++ programs;

— the restrictions and limits imposed by a conforming implementation ~~of C~~to be successfully translated in C or C++ environments.

2 This document does not specify

— the mechanism by which ~~C~~ programs are transformed for use by a data-processing system;

— the mechanism by which ~~C~~ programs are invoked for use by a data-processing system;

— the mechanism by which input data are transformed for use by a ~~C~~ program;

— the mechanism by which output data are transformed after being produced by a ~~C~~ program;

— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;

— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

[3]This document is designed to promote the portability of C and C++ programs among a variety of data-processing systems. It is intended for use by implementors and programmers. Annex J gives an overview of portability issues that a program might encounter.

2. Normative references

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2 ISO/IEC 2382:2015, *Information technology — Vocabulary*. Available from the *ISO online browsing platform* at `http://www.iso.org/obp`.

3 ISO 4217, *Codes for the representation of currencies and funds*.

4 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

5 ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*. Available from the ISO/IEC Information Technology Task Force (ITTF) web site at `http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm`.

6 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989).

7 ISO 80000–2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*.

8 *Also add* ISO 80000–3, space and time

9 *Also add* ISO 80000–13, Information science and technology

3. Terms, definitions, and symbols

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000–2, and the following apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at `https://www.iso.org/obp`

— IEC Electropedia: available at `http://www.electropedia.org/`

3 Additional terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere.

3.1

1 **access (verb)**

⟨execution-time action⟩ to read or modify the value of an object

2 **Note 1 to entry:** Where only one of these two actions is meant, "read" or "modify" is used.

3 **Note 2 to entry:** "Modify" includes the case where the new value being stored is the same as the previous value.

4 **Note 3 to entry:** Expressions that are not evaluated do not access objects.

3.2

1 **alignment**

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.3

1 **argument**

actual argument

DEPRECATED: actual parameter

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4

1 **behavior**

external appearance or action

3.4.1

1 **implementation-defined behavior**

unspecified behavior where each implementation documents how the choice is made

2 **Note 1 to entry:** J.3 gives an overview over properties of C programs that lead to implementation-defined behavior.

3 **EXAMPLE** An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2

1 **locale-specific behavior**

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

2 **Note 1 to entry:** J.4 gives an overview over properties of C programs that lead to locale-specific behavior.

3 **EXAMPLE** An example of locale-specific behavior is whether the `islower` function returns true for characters other than the 26 lowercase Latin letters.

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes no requirements

2 **Note 1 to entry:** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 **Note 2 to entry:** J.2 gives an overview over properties of C programs that lead to undefined behavior.

4 **EXAMPLE** An example of undefined behavior is the behavior on integer overflow.

3.4.4

1 **unspecified behavior**

behavior, that results from the use of an unspecified value, or other behavior upon which this document provides two or more possibilities and imposes no further requirements on which is chosen in any instance

2 **Note 1 to entry:** J.1 gives an overview over properties of C programs that lead to unspecified behavior.

3 **EXAMPLE** An example of unspecified behavior is the order in which the arguments to a function are evaluated.

3.5

1 **bit**

unit of data storage in the execution environment large enough to hold an object that can have one of two values

2 **Note 1 to entry:** It need not be possible to express the address of each individual bit of an object.

3.6

1 **byte**

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

2 **Note 1 to entry:** It is possible to express the address of each individual byte of an object uniquely.

3 **Note 2 to entry:** A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

3.7

1 **character**

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

3.7.1

1 **character**

single-byte character

⟨C⟩ bit representation that fits in a byte

3.7.2

1 **multibyte character**

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

2 **Note 1 to entry:** The extended character set is a superset of the basic character set.

3.7.3

1 **wide character**

value representable by an object of type `wchar_t`, capable of representing any character in the current locale

3.8

1 **constraint**

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

3.9

1 **correctly rounded result**

representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

2 **Note 1 to entry:** In this document, when the words "correctly rounded" are not immediately followed by "result", this is the intended usage.

3.10

1 **diagnostic message**

message belonging to an implementation-defined subset of the implementation's message output

3.11

1 **forward reference**

reference to a later subclause of this document that contains additional information relevant to this subclause

3.12

1 **implementation**

particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13

1 **implementation limit**

restriction imposed upon programs by the implementation

3.14

1 **memory location**

either an object of scalar type, or a ~~maximal sequence of adjacent bit-fields all having nonzero width~~ pack

2 **Note 1 to entry:** Two threads of execution can update and access separate memory locations without interfering with each other.

3 **Note 2 to entry:** A ~~bit-field~~ member with `core::alias` attribute and an adjacent ~~non-bit-field member~~ member without are in separate memory locations. The same applies to two ~~bit-fields~~such members with the attribute, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a ~~non-bit-field member declaration~~ member declaration without the attribute. It is not safe to concurrently update two such non-atomic ~~bit-fields~~ members if they are in the same ~~structure if all members declared between them are also (nonzero-length) bit-fields~~pack, no matter what the sizes of those intervening bit-fields happen to be.

4 **EXAMPLE** A structure declared as

```
struct {
        char a;
        int b:5, c:11,:0, d:8;
```

```
            struct { int ee:8; } e;
            signed b:5, c:11,:0, d:8;
            struct { [[core::alias]] signed char ee; } e;
      }
```

contains four separate memory locations: The ~~member `a`, and bit-fields~~ members `a`, `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be.

3.15

1 **object**

region of data storage in the execution environment, the contents of which can represent values

2 **Note 1 to entry:** When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **pack**

a maximal sequence of adjacent members that have the **`core::alias`** attribute

3.17

1 **parameter**

formal parameter

DEPRECATED: formal argument

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.18

1 **pointer provenance**

provenance

storage instance that holds the object to which a valid pointer value refers

3.19

1 **recommended practice**

specification that is strongly recommended as being in keeping with the intent of the standard, but that might be impractical for some implementations

3.20

1 **runtime-constraint**

requirement on a program when calling a library function

2 **Note 1 to entry:** Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 **Note 2 to entry:** Implementations that support ~~the extensions in ?? are required to verify that the runtime-constraints for a library function are not violated by the program; see ??. Implementations that support~~ Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.21

1 **storage instance**

the inclusion-maximal region of data storage in the execution environment that is created when either an object definition or an allocation is encountered

2 **Note 1 to entry:** Storage instances are created and destroyed when specific language constructs (6.2.4) are met during program execution, including program startup, or when specific library functions (7.22.4) are called.

3 **Note 2 to entry:** A given storage instance may or may not have a memory address, and may or may not be accessible from

all threads of execution.

4 **Note 3 to entry:** Storage instances have identities which are unique across the program execution.

5 **Note 4 to entry:** A storage instance with a memory address occupies a region of zero or more bytes of contiguous data storage in the execution environment.

6 **Note 5 to entry:** One or more objects may be represented within the same storage instance, such as two subobjects within an object of structure type, two `const`-qualified compound literals with identical object representation, or two string literals where one is the terminal character sequence of the other.

3.22

1 **value**

precise meaning of the contents of an object when interpreted as having a specific type

3.22.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.22.2

1 **indeterminate value**

either an unspecified value or a trap representation

3.22.3

1 **unspecified value**

valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance

2 **Note 1 to entry:** An unspecified value cannot be a trap representation.

3.22.4

1 **trap representation**

an object representation that need not represent a value of the object type

3.22.5

1 **perform a trap**

interrupt execution of the program such that no further operations are performed

2 **Note 1 to entry:** In this document, when the word "trap" is not immediately followed by "representation", this is the intended usage.[4)]

3 **Note 2 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.23

1 $\lceil x \rceil$

ceiling of x

the least integer greater than or equal to x

2 **EXAMPLE** $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2.

3.24

1 $\lfloor x \rfloor$

floor of x

the greatest integer less than or equal to x

2 **EXAMPLE** $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3.

[4)]For example, "Trapping or stopping (if supported) is disabled . . . " (F.8.2). Note that fetching a trap representation might perform a trap but is not required to (see 6.2.6.1).

4. Conformance

1 In this document, "shall" is to be interpreted as a requirement on an implementation or on a program; conversely, "shall not" is to be interpreted as a prohibition.

2 If a "shall" or "shall not" requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this document by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined".

3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

4 The implementation shall not successfully translate a preprocessing translation unit containing a **#error** preprocessing directive unless it is part of a group skipped by conditional inclusion.

5 A *strictly conforming program* shall use only those features of the language and library specified in this document.[5] It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

6 The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library clause (Clause 7) is confined to the contents of the standard headers <float.h>, ~~,~~<limits.h>, ~~,~~ <stdarg.h>, ~~, ,~~<stdint.h>, and <stdnoreturn.h>.[6] A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[7]

7 A *conforming program* is one that is acceptable to a conforming implementation.[8]

8 An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types <float.h> (7.7), alternative spellings <iso646.h> (7.9), sizes of integer types <limits.h> (7.10), alignment <stdalign.h> (7.15), variable arguments <stdarg.h> (7.16), boolean type and values <stdbool.h> (7.18), common definitions <stddef.h> (7.19), integer types <stdint.h> (7.20), <stdnoreturn.h> (7.23).

[5]A strictly conforming program can use conditional features (see 6.10.8.2) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```
#ifdef __STDC_IEC_559__ /* FE_UPWARD defined */
   /* ... */
   fesetround(FE_UPWARD);
   /* ... */
#endif
```

[6]The features that historically had been presented by the headers <iso646.h>, <stdalign.h>, <stdbool.h>, and <stddef.h> are properly integrated into the C/C++ core and do not need to be present as separate headers.

[7]This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this document.

[8]Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs can depend upon nonportable features of a conforming implementation.

5. Environment

1 An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this document. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In this clause, only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

1 A C program need not all be translated at the same time. The text of the program is kept in units called *source files*, (or *preprocessing files*) in this document. A source file together with all the headers and source files included via the preprocessing directive **`#include`** is known as a *preprocessing translation unit*. After preprocessing, a preprocessing translation unit is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

1 The precedence among the syntax rules of translation is specified by the following phases.[9]

1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary.~~Trigraph sequences are replaced by corresponding single-character internal representations.~~[10] Any source file character not in the basic source character set is replaced by the universal character name that designates that character. An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal character name (*e.g.*, using the `\uXXXX` notation), are handled equivalently.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. If a splice results in a character sequence that matches the syntax of a universal character name, the behavior is undefined. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

3. The source file is decomposed into preprocessing tokens[11] and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters

[9]This requires implementations to behave as if these separate phases occur, even though many are typically folded together in practice. Source files, translation units, and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

[10]Historically, in this phase also trigraph sequences would have been replaced by corresponding single-character internal representations, see 5.2.1.1.

[11]As described in 6.4, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of < within a **`#include`** preprocessing directive.

are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed, macro invocations are expanded, and **`_Pragma`** unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. A **`#include`** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each source character set member~~and escape sequence~~ , escape sequence and universal character name in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[12)]

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), ~~trigraph sequences (??),~~ external definitions (6.9).

5.1.1.3 Diagnostics

1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.[13)]

2 **EXAMPLE** An implementation is required to issue a diagnostic for the translation unit:

```
char i;
int i;
```

because in those cases where wording in this document describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error is still required to be diagnosed.

5.1.2 Execution environments

1 Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects with static storage duration shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

Forward references: storage durations of objects (6.2.4), initialization (6.7.12).

5.1.2.1 Freestanding environment

1 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by Clause 4, are implementation-defined.

12) An implementation need not convert all non-corresponding source characters to the same execution character.

13) An implementation is encouraged to identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostic messages, often referred to as warnings, as long as a valid program is still correctly translated. It can also successfully translate an invalid program. Annex I lists a few of the more common warnings.

2 The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

1 A hosted environment need not be provided, but shall conform to the following specifications if present.

5.1.2.2.1 Program startup

1 The function called at program startup is named `main`. The implementation declares no prototype for this function. It shall be defined with a return type of `int` and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[]) { /* ... */ }
```

or equivalent;[14] or in some other implementation-defined manner.

2 If they are declared, the parameters to the `main` function shall obey the following constraints:

— The value of `argc` shall be nonnegative.

— `argv[argc]` shall be a null pointer.

— If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

— If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the *program name*; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the *program parameters*.

— The parameters `argc` and `argv` and the strings pointed to by the `argv` array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

5.1.2.2.2 Program execution

1 In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (Clause 7).

5.1.2.2.3 Program termination

1 If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument;[15] reaching the `}` that terminates the `main` function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

Forward references: definition of terms (7.1.1), the `exit` function (7.22.5.4).

[14] Thus, `int` can be replaced by a typedef name defined as `int`, or the type of `argv` can be written as `char ** argv`, and so on.

[15] In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in `main` will have ended in the former case, even where they would not have in the latter.

5.1.2.3 Program execution

1 The semantic descriptions in this document describe the behavior of an abstract machine in which issues of optimization are irrelevant.

2 An access to an object through the use of an lvalue of volatile-qualified type is a *volatile access*. A volatile access to an object, modifying a file, or calling a function that does any of those operations are all *side effects*,[16)] which are changes in the state of the execution environment. *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

3 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is *sequenced after* A.) If A is not sequenced before or after B, then A and B are *unsequenced*. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which.[17)] The presence of a *sequence point* between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the sequence points is given in Annex C.)

4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or through volatile access to an object).

5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **`volatile`** `sig_atomic_t` are unspecified, as is the state of the floating-point environment. The value of any object modified by the handler that is neither a lock-free atomic object nor of type **`volatile`** `sig_atomic_t` becomes indeterminate when the handler exits, as does the state of the floating-point environment if it is modified by the handler and not restored to its original state.

6 The least requirements on a conforming implementation are:

— Volatile accesses to objects are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

This is the *observable behavior* of the program.

7 What constitutes an interactive device is implementation-defined.

8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.

9 **EXAMPLE 1** An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **`volatile`** would then be redundant.

10 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In

[16)]The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see Annex F for details. The floating-point environment library `<fenv.h>` provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

[17)]The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the `signal` function would require explicit specification of `volatile` storage, as well as other implementation-defined restrictions.

11 **EXAMPLE 2** In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the "integer promotions" require that the abstract machine promote the value of each variable to `int` size and then add the two `int`s and truncate the sum. Provided the addition of two `char`s can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

12 **EXAMPLE 3** Similarly, in the fragment

```
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
f1 = f2 × d;
```

the multiplication can be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if `d` were replaced by the constant `2.0`, which has type `double`).

13 **EXAMPLE 4** Implementations employing wide hardware registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a hardware register or in memory. For example, an implicit *spilling* of a hardware register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to `d1` and `d2` are required to have been converted to `float`.

14 **EXAMPLE 5** Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.9).

```
double x, y, z;
/* ... */
x = (x * y) * z;  // not equivalent to x *= y * z;
x = (x × y) × z;  // not equivalent to x ×= y × z;
z = (x - y) + y;  // not equivalent to z = x;
z = x + x * y;    // not equivalent to z = x * (1.0 + y);
y = x / 5.0;      // not equivalent to y = x * 0.2;
z = x + x × y;    // not equivalent to z = x × (1.0 + y);
y = x / 5.0;      // not equivalent to y = x × 0.2;
```

15 **EXAMPLE 6** To illustrate the grouping behavior of expressions, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum `(a + 32760)` is next added to `b`, and that result is then added to 5 which results in the value assigned to `a`. On a machine in which overflows produce an explicit trap and in which the range of values representable by an **int** is $[-32768, +32767]$, the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for `a` and `b` were, respectively, -32754 and -15, the sum `a + b` would produce a trap while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

16 **EXAMPLE 7** The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
sum = sum × 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
sum = (((sum × 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of `p` can occur at any time between the previous sequence point and the next sequence point (the `;`), and the call to **getchar** can occur at any point prior to the need of its returned value.

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment `<fenv.h>` (7.6), the **signal** function (7.14), files (7.21.3).

5.1.2.4 Multi-threaded executions and data races

1 Under a hosted implementation, a program can have more than one *thread of execution* (or *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads.[18] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 The value of an object visible to a thread T at a particular point is the initial value of the object, a value stored in the object by T, or a value stored in the object by another thread, according to the rules below.

3 **NOTE 1** In some cases, there could instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs.

4 Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.

5 ~~The library defines~~ There are a number of ~~(7.17) and~~ operations that are specially identified as synchronization operations: these are operators and generic functions (if the implementation supports the atomics extension) that act on atomic objects (6.5 and 7.17); if the implementation supports the thread extension these are calls to initialization functions (7.26.2), operations on mutexes ~~(7.26.4) that are specially identified as synchronization operations.~~ 7.26.3 and 7.26.4), and calls

[18]The execution can usually be viewed as an interleaving of all of the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below.

to thread functions (7.26.5). These operations play a special role in making ~~assignments~~ side effects in one thread visible to another. A *synchronization operation* on one or more memory locations is either an *acquire operation*, a *release operation*, both an acquire and release operation, or a *consume operation*. A synchronization operation without an associated memory location is a *fence* and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are *relaxed atomic operations*, which are not synchronization operations but still are indivisible, and atomic *read-modify-write operations*, which ~~have special characteristics.~~ are those operations defined in 6.5 and 7.17 that act on an atomic object by reading its value, by performing an optional operation with that value and by storing back a value into that object.

6 **NOTE 2** For example, a call that acquires a mutex will perform an acquire operation on the locations composing the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform an acquire or consume operation on A. Relaxed atomic operations are not included as synchronization operations although, like synchronization operations, they cannot contribute to data races.

7 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M, and A happens before B, then A shall precede B in the modification order of M, which is defined below.

8 **NOTE 3** This states that the modification orders are expected to respect the "happens before" relation.

9 **NOTE 4** There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different variables in inconsistent orders.

10 A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A and every subsequent operation either is performed by the same thread that performed the release or is an atomic read-modify-write operation.

11 Certain ~~library calls~~ operations *synchronize with* other ~~library calls~~ operations performed by another thread. In particular, an atomic operation A that performs a release operation on an object M synchronizes with an atomic operation B that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by A.

12 **NOTE 5** Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation.

13 **NOTE 6** The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release.

14 An evaluation A *carries a dependency*[19] to an evaluation B if:

— the value of A is used as an operand of B, unless:

- B is an invocation of the **`kill_dependency`** macro,
- A is the left operand of a ~~&&~~ $\wedge$ or ~~|| operator,~~ $\vee$ operator,
- A is the left operand of a `?:` operator, or
- A is the left operand of a `,` operator;

or

— A writes a scalar object or ~~bit-field~~ **`core::alias`** member M, B reads from M the value written by A, and A is sequenced before B, or

— for some evaluation X, A carries a dependency to X and X carries a dependency to B.

15 An evaluation A is *dependency-ordered before*[20] an evaluation B if:

[19] The "carries a dependency" relation is a subset of the "sequenced before" relation, and is similarly strictly intra-thread.

[20] The "dependency-ordered before" relation is analogous to the "synchronizes with" relation, but uses release/consume in place of release/acquire.

— A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

— for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

16 An evaluation A *inter-thread happens before* an evaluation B if A synchronizes with B, A is dependency-ordered before B, or, for some evaluation X:

— A synchronizes with X and X is sequenced before B,

— A is sequenced before X and X inter-thread happens before B, or

— A inter-thread happens before X and X inter-thread happens before B.

17 **NOTE 7** The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with", and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before".

18 An evaluation A *happens before* an evaluation B if A is sequenced before B or A inter-thread happens before B. The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation.

19 **NOTE 8** This cycle would otherwise be possible only through the use of consume operations.

20 A *visible side effect* A on an object M with respect to a value computation B of M satisfies the conditions:

— A happens before B, and

— there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object M, as determined by evaluation B, shall be the value stored by the visible side effect A.

21 **NOTE 9** If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.

22 **NOTE 10** This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.

23 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

24 **NOTE 11** The set of side effects from which a given evaluation might take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below.

25 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M.

26 **NOTE 12** The requirement above is known as "write-write coherence".

27 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M.

28 **NOTE 13** The requirement above is known as "read-read coherence".

29 If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.

30 **NOTE 14** The requirement above is known as "read-write coherence".

31 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

32 **NOTE 15** The requirement above is known as "write-read coherence".

33 **NOTE 16** This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are "relaxed" loads. By doing so, it effectively makes the "cache coherence" guarantee provided by most hardware available to C atomic operations.

34 **NOTE 17** The value observed by a load of an atomic object depends on the "happens before" relation, which in turn depends on the values observed by loads of atomic objects. The intended reading is that there exists an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here.

35 Two evaluations are *concurrent* if neither happens before the other. The execution of a program contains a *data race* if it contains two concurrent conflicting actions in different threads, at least one of which is not atomic ~~, and neither happens before the other.~~ or if they access an atomic object that has not been initialized. Any such data race results in undefined behavior.

36 **NOTE 18** It can be shown that programs that correctly use simple mutexes and `memory_order_seq_cst` operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result necessarily has undefined behavior even before such a transformation is applied.

37 **NOTE 19** Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence requirements.

38 **NOTE 20** Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

1 Two sets of characters and their associated ~~collating sequences~~ *collating sequences* shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. The combined set is also called the *extended character set*. The values of the members of the execution character set are implementation-defined.

2 In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash `\` followed by one or more characters. A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string.

Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the *Latin alphabet*

3
```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

the 26 *lowercase letters* of the Latin alphabet

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

the 10 decimal *digits*

```
0 1 2 3 4 5 6 7 8 9
```

the following 29 *graphic characters*

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed. The representation of each member of the source and execution basic character sets shall fit in a byte. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this document treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined.

4 A *letter* is an uppercase letter or a lowercase letter as defined above; in this document the term does not include other characters that are letters in other alphabets.

5 The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1).

~~Before any other processing takes place, each occurrence of one of the following sequences of three characters (called)[21] is replaced with the corresponding single character.~~

~~??= #~~
~~??([~~
~~??/ \~~
~~??)]~~
~~??' ^~~
~~??< {~~
~~??! |~~
~~??> }~~
~~??- ~~~

~~No other trigraph sequences exist. Each ? that does not begin one of the trigraphs listed above is not changed. becomes The following source line becomes (after replacement of the trigraph sequence ??/)~~

5.2.1.1 Multibyte characters

1 The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:

— The basic character set shall be present and each character shall be encoded as a single byte.

— The presence, meaning, and representation of any additional members is locale-specific.

— A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

[21] ~~The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set.~~

— A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.

2 For source files, the following shall hold:

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

3 **NOTE** Historically, C and C++ also had *trigraph sequences* , such that all occurences of the following triplets where replaced with the corresponding single character during translation phase 1.

```
??=  #        ??)  ]        ??!  |
??(  [        ??'  ^        ??>  }
??/  \        ??<  {        ??-  ~
```

C++ has now completely removed them, and also some C compilers only support these with additional commandline options. They are only marginally used nowadays and therefore removed from the C/C++ core.

5.2.2 Character display semantics

1 The *active position* is that location on a display device where the next character output by the `fputc` function would appear. The intent of writing a printing character (as defined by the `isprint` function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

2 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:

`\a` (*alert*) Produces an audible or visible alert without changing the active position.

`\b` (*backspace*) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.

`\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page.

`\n` (*new line*) Moves the active position to the initial position of the next line.

`\r` (*carriage return*) Moves the active position to the initial position of the current line.

`\t` (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.

`\v` (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

3 Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single **`char`** object. The external representations in a text file need not be identical to the internal representations, and are outside the scope of this document.

Forward references: the `isprint` function (7.4.1.8), the `fputc` function (7.21.7.3).

5.2.3 Signals and interrupts

1 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis.

5.2.4 Environmental limits

1 Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the language-related environmental limits on a conforming implementation; the library-related limits are discussed in Clause 7.

5.2.4.1 Translation limits

1 The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:[21)]

— 127 nesting levels of blocks

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or **`void`** type in a declaration

— 63 nesting levels of parenthesized declarators within a full declarator

— 63 nesting levels of parenthesized expressions within a full expression

— 63 significant initial characters in an internal identifier or a macro name(each universal character name or extended source character is considered a single character)

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[22)]

— 4095 external identifiers in one translation unit

— 511 identifiers with block scope declared in one block

— 4095 macro identifiers simultaneously defined in one preprocessing translation unit

— 127 parameters in one function definition

— 127 arguments in one function call

— 127 parameters in one macro definition

— 127 arguments in one macro invocation

— 4095 characters in a logical source line

— 4095 characters in a string literal (after concatenation)

— 65535 bytes in an object (in a hosted environment only)

— 15 nesting levels for **`#include`**d files

— 1023 **`case`** labels for a **`switch`** statement (excluding those for any nested **`switch`** statements)

— 1023 members in a single structure or union

— 1023 enumeration constants in a single enumeration

— 63 levels of nested structure or union definitions in a single member declaration list

2 The implementation shall be able to translate constant expressions that do not exceed the following limits. Other implementation-defined limits may be specified that constrain the evaluation of constant expressions and possible calls to **`constexpr`** functions in such a context.

[21)] Implementations are encouraged to avoid imposing fixed translation limits whenever possible.
[22)] See "future language directions" (6.11.3).

— 512 recursive **constexpr** function invocations

— 1048576 full-expressions evaluated within a constant expression

5.2.4.2 Numerical limits

1 An implementation is required to document all the limits specified in this subclause, which are specified in the headers `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`.

Forward references: integer types `<stdint.h>` (7.20).

5.2.4.2.1 Characteristics of integer types `<limits.h>`

1 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Their implementation-defined values shall be equal or greater to those shown.

— width for an object of type **_Bool**

BOOL_WIDTH	1

— number of bits for smallest object ~~that is not a bit-field~~ (byte)

CHAR_BIT	8

The macros **CHAR_WIDTH**, **SCHAR_WIDTH**, and **UCHAR_WIDTH** that represent the width of the types **char**, **signed char** and **unsigned char** shall expand to the same value as **CHAR_BIT**.

— width for an object of type **unsigned short int**

USHRT_WIDTH	16

The macro **SHRT_WIDTH** represents the width of the type **short int** and shall expand to the same value as **USHRT_WIDTH**.

— width for an object of type **unsigned int**

UINT_WIDTH	16

The macro **INT_WIDTH** represents the width of the type **int** and shall expand to the same value as **UINT_WIDTH**.

— width for an object of type **unsigned long int**

ULONG_WIDTH	32

The macro **LONG_WIDTH** represents the width of the type **long int** and shall expand to the same value as **ULONG_WIDTH**.

— width for an object of type **unsigned long long int**

ULLONG_WIDTH	64

The macro **LLONG_WIDTH** represents the width of the type **long long int** and shall expand to the same value as **ULLONG_WIDTH**.

— the maximum width for a bit-set

INT_BITFIELD_MAX	UINT_WIDTH

— maximum number of bytes in a multibyte character, for any supported locale

`MB_LEN_MAX`	1

2 For all unsigned integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix `_WIDTH` holding its width N, there is a macro with suffix `_MAX` holding the maximal value $2^N - 1$ that is representable by the type, that is suitable for use in **`#if`** preprocessing directives and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

3 For all signed integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix `_WIDTH` holding its width N, there are macros with suffix `_MIN` and `_MAX` holding the minimal and maximal values -2^{N-1} and $2^{N-1} - 1$ that are representable by the type, that are suitable for use in **`#if`** preprocessing directives and that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

4 If an object of type **`char`** can hold negative values, the value of `CHAR_MIN` shall be the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` shall be the same as that of `SCHAR_MAX`. Otherwise, the value of `CHAR_MIN` shall be 0 and the value of `CHAR_MAX` shall be the same as that of `UCHAR_MAX`.[23)]

Forward references: representations of types (6.2.6), conditional inclusion (6.10.1), integer types `<stdint.h>` (7.20).

5.2.4.2.2 Characteristics of floating types `<float.h>`

1 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[24)] An implementation that defines `__STDC_IEC_559__` shall implement floating point types and arithmetic conforming to IEC 60559 as specified in Annex F. ~~An implementation that defines `__STDC_IEC_559_COMPLEX__` shall implement complex types and arithmetic conforming to IEC 60559 as specified in ??.~~

2 The following parameters are used to define the model for each floating-point type:

s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

For each floating-point type the parameters b, p, $e_{\min}$, and $e_{\max}$, are fixed constants.

3 For each floating-point type, a *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

4 Floating types shall be able to represent zero (all $\cancel{f_k == 0}$ $f_k \equiv 0$) and all *normalized floating-point numbers* ($f_1 > 0$ and all possible k digits and e exponents result in values representable in the type). In addition, floating types may be able to contain other kinds of floating-point numbers,[25)] such as *negative zero*, *subnormal floating-point numbers* ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$) and *unnormalized floating-point numbers* ($x \neq 0$, $e > e_{\min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.[26)]

5 An implementation may give zero and values that are not floating-point numbers (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement

[23)] See 6.2.5.

[24)] The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical.

[25)] Some implementations have types that include finite numbers with extra range and/or precision that are not covered by the model.

[26)] IEC 60559:1989 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

in this document to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

6 The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. In addition, if negative infinity is representable in a type, the range of that type is extended to all negative real numbers; likewise, if positive infinity is representable in a type, the range of that type is extended to all positive real numbers.

7 The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown.

8 All integer values in the `<float.h>` header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types. The floating-point model representation is provided for all values except **FLT_EVAL_METHOD** and **FLT_ROUNDS**.

9 The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**. Evaluation of **FLT_ROUNDS** correctly reflects any execution-time change of rounding mode through the function **fesetround** in `<fenv.h>`.

−1	indeterminable
0	toward zero
1	to nearest, ties to even
2	toward positive infinity
3	toward negative infinity
4	to nearest, ties away from zero

All other values for **FLT_ROUNDS** characterize implementation-defined rounding behavior.

10 The values of floating type yielded by operators subject to the usual arithmetic conversions, including the values yielded by the implicit conversion of operands, and the values of floating constants are evaluated to a format whose range and precision may be greater than required by the type. Such a format is called an *evaluation format*. In all cases, assignment and cast operators yield values in the format of the type. The extent to which evaluation formats are used is characterized by the value of **FLT_EVAL_METHOD**:[27)]

−1	indeterminable;
0	evaluate all operations and constants just to the range and precision of the type;
1	evaluate operations and constants of type **float** and **double** to the range and precision of the **double** type, evaluate **long double** operations and constants to the range and precision of the **long double** type;
2	evaluate all operations and constants to the range and precision of the **long double** type.

All other negative values for **FLT_EVAL_METHOD** characterize implementation-defined behavior. The value of **FLT_EVAL_METHOD** does not characterize values returned by function calls (see 6.8.6.4, F.6).

11 The presence or absence of subnormal numbers is characterized by the implementation-defined values of **FLT_HAS_SUBNORM**, **DBL_HAS_SUBNORM**, and **LDBL_HAS_SUBNORM**:

[27)]The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **FLT_EVAL_METHOD** is 1, then the product of two **complex_type**(**float**) operands is represented in the **complex_type**(**double**) format, and its parts are evaluated to **double**.

-1 indeterminable[28]

0 absent (type does not support subnormal numbers)[29]

1 present (type does support subnormal numbers)

12 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

— radix of exponent representation, b

```
FLT_RADIX                         2
```

— number of base-**FLT_RADIX** digits in the floating-point significand, p

```
FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG
```

— number of decimal digits, n, such that any floating-point number with p radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
FLT_DECIMAL_DIG                    6
DBL_DECIMAL_DIG                   10
LDBL_DECIMAL_DIG                  10
```

— number of decimal digits, n, such that any floating-point number in the widest supported floating type with $p_{\max}$ radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p_{\max} \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
DECIMAL_DIG                       10
```

This is an obsolescent feature, see 7.32.6.

— number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

```
FLT_DIG                            6
DBL_DIG                           10
LDBL_DIG                          10
```

[28]Characterization as indeterminable is intended if floating-point operations do not consistently interpret subnormal representations as zero, nor as nonzero.

[29]Characterization as absent is intended if no floating-point operations produce subnormal results from non-subnormal inputs, even if the type format includes representations of subnormal numbers.

— minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, $e_{\min}$

```
FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP
```

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil log_{10} b^{e_{\min}-1} \rceil$

```
FLT_MIN_10_EXP                -37
DBL_MIN_10_EXP                -37
LDBL_MIN_10_EXP               -37
```

— maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, $e_{\max}$

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor log_{10}((1 - b^{-p})b^{e_{\max}}) \rfloor$

```
FLT_MAX_10_EXP                +37
DBL_MAX_10_EXP                +37
LDBL_MAX_10_EXP               +37
```

13 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown:

— maximum representable finite floating-point number; if that number is normalized, its value is $(1 - b^{-p})b^{e_{\max}}$

```
FLT_MAX                     1E+37
DBL_MAX                     1E+37
LDBL_MAX                    1E+37
```

— maximum normalized floating-point number, $(1 - b^{-p})b^{e_{\max}}$

```
FLT_NORM_MAX                1E+37
DBL_NORM_MAX                1E+37
LDBL_NORM_MAX               1E+37
```

14 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

— the difference between 1 and the least normalized value greater than 1 that is representable in the given floating-point type, b^{1-p}

```
FLT_EPSILON                  1E-5
DBL_EPSILON                  1E-9
LDBL_EPSILON                 1E-9
```

— minimum normalized positive floating-point number, $b^{e_{\min}-1}$

```
FLT_MIN                  1E-37
DBL_MIN                  1E-37
LDBL_MIN                 1E-37
```

— minimum positive floating-point number[30)]

```
FLT_TRUE_MIN             1E-37
DBL_TRUE_MIN             1E-37
LDBL_TRUE_MIN            1E-37
```

Recommended practice

15 Conversion between real floating type and decimal character sequence with at most *T*`_DECIMAL_DIG` digits should be correctly rounded, where *T* is the macro prefix for the type. This assures conversion from real floating type to decimal character sequence with *T*`_DECIMAL_DIG` digits and back, using to-nearest rounding, is the identity function.

16 **EXAMPLE 1** The following describes an artificial floating-point representation that meets the minimum requirements of this document, and the appropriate values in a `<float.h>` header for type **`float`**:

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
    FLT_RADIX                       16
    FLT_MANT_DIG                     6
    FLT_EPSILON           9.53674316E-07F
    FLT_DECIMAL_DIG                  9
    FLT_DIG                          6
    FLT_MIN_EXP                    -31
    FLT_MIN               2.93873588E-39F
    FLT_MIN_10_EXP                 -38
    FLT_MAX_EXP                    +32
    FLT_MAX               3.40282347E+38F
    FLT_MAX_10_EXP                 +38
```

17 **EXAMPLE 2** The following describes floating-point representations that also meet the requirements for single-precision and double-precision numbers in IEC 60559,[31)] and the appropriate values in a `<float.h>` header for types **`float`** and **`double`**:

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
    FLT_RADIX                        2
    FLT_MANT_DIG                    24
    FLT_EPSILON           1.19209290E-07F // decimal constant
    FLT_EPSILON                  0X1P-23F // hex constant
    FLT_DECIMAL_DIG                  9
    FLT_DIG                          6
    FLT_MIN_EXP                   -125
    FLT_MIN               1.17549435E-38F // decimal constant
    FLT_MIN                     0X1P-126F // hex constant
    FLT_TRUE_MIN          1.40129846E-45F // decimal constant
    FLT_TRUE_MIN                0X1P-149F // hex constant
    FLT_HAS_SUBNORM                  1
```

[30)] If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.

[31)] The floating-point model in that standard sums powers of b from zero, so the values of the exponent limits are one less than shown here.

```
    FLT_MIN_10_EXP                  -37
    FLT_MAX_EXP                    +128
    FLT_MAX             3.40282347E+38F // decimal constant
    FLT_MAX             0X1.fffffeP127F // hex constant
    FLT_MAX_10_EXP                  +38
    DBL_MANT_DIG                     53
    DBL_EPSILON   2.2204460492503131E-16 // decimal constant
    DBL_EPSILON                 0X1P-52 // hex constant
    DBL_DECIMAL_DIG                  17
    DBL_DIG                          15
    DBL_MIN_EXP                   -1021
    DBL_MIN      2.2250738585072014E-308 // decimal constant
    DBL_MIN                   0X1P-1022 // hex constant
    DBL_TRUE_MIN 4.9406564584124654E-324 // decimal constant
    DBL_TRUE_MIN              0X1P-1074 // hex constant
    DBL_HAS_SUBNORM                   1
    DBL_MIN_10_EXP                 -307
    DBL_MAX_EXP                   +1024
    DBL_MAX      1.7976931348623157E+308 // decimal constant
    DBL_MAX       0X1.fffffffffffffP1023 // hex constant
    DBL_MAX_10_EXP                 +308
```

Forward references: conditional inclusion (6.10.1), predefined macro names (6.10.8), complex arithmetic `<complex.h>` (7.3), extended multibyte and wide character utilities `<wchar.h>` (7.29), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.22), input/output `<stdio.h>` (7.21), mathematics `<math.h>` (7.12), IEC 60559 floating-point arithmetic (Annex F~~), IEC 60559-compatible complex arithmetic (??~~).

6. Language

6.1 Notation

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

{ $expression_{\text{opt}}$ **}**

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

3 A label name is the only kind of identifier that has *function scope*. It can be used (in a `goto` statement) anywhere in the ~~function~~ innermost function body in which it appears, ~~and~~ but not from within another function body of a lambda expression that is evalutated within that function body.[32] A label is declared implicitly by its syntactic appearance (followed by a `:` and a statement).

4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block~~or list of parameters~~, capture or parameter list, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block, or within the ~~list of parameter declarations in~~ parameter list of a function definition ~~,~~ or lambda expression, or if the identifier is a capture the identifier has *block scope*, which terminates at the end of the associated block or function body. If the declarator or type specifier that declares the identifier appears within the ~~list of parameter declarations in~~ parameter list of a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

5 Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

[32] So labels can not be used to jump in our out of lambda expressions.

6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. An identifier that has an underspecified declarator and that designates an object has a scope that starts at the end of its initializer; if the same identifier declares another entity in an surrounding scope, that declaration is hidden as soon as the inner declarator is met.[33] An identifier that designates a function with an underspecified return type has a scope that starts after its first **`return`** statement if there is one, or at the end of the function body if there is none, and from that point extends to the whole translation unit. Any other identifier has scope that begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[34] There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[35]

4 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[36] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **`extern`**. If the declaration of an identifier for an object has file scope and no storage-class specifier or only the specifier **`auto`** , its linkage is external, unless it has a lambda value, in which case it has internal linkage.

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **`extern`**.

7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, in translation phase 7 there are separate *name spaces* for various categories of identifiers, as follows:

[33] That means, that the outer declaration is not visible for the initializer.

[34] There is no linkage between different identifiers.

[35] A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1.

[36] As specified in 6.2.1, the later declaration might hide the prior declaration.

— *label names* (disambiguated by the syntax of the label declaration and use);

— the *tags* of structures, unions, and enumerations (disambiguated by following any[37] of the keywords **`struct`**, **`union`**, or **`enum`**);

— the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or ~~`->`~~ → operator);

— standard attributes and attribute prefixes (disambiguated by the syntax of the attribute specifier and the name of the attribute token) (6.7.15);

— the *attribute suffixes* of an attribute prefixed token; each attribute prefix has a separate name space for the implementation-defined attributes that it introduces (disambiguated by the attribute prefix and the :: token);

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), attributes (6.7.15), the **`goto`** statement (6.8.6.1).

~~An object has a that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in **??**.~~

6.2.4 Storage durations and object lifetimes

1 The *lifetime* of an object ~~is the portion of program execution during which storage is guaranteed~~ has a start and an end, which both constitute side effects in the abstract state machine, and is the set of all evaluations that happen after the start and before the end. An object exists, has a storage instance that is guaranteed to be reserved for it~~. An object exists~~, has a constant address,[38] if any, and retains its last-stored value throughout its lifetime.[39] ~~If~~

2 The lifetime of an object is ~~referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime~~ determined by its *storage duration* . There are four storage durations: static, thread, automatic, and allocated. Allocated storage and its duration are described in 6.7.15.3 and 7.22.4.

3 Object definitions (6.7) do not have allocated storage duration and give rise to a unique storage instance that has the same lifetime as the object that is defined. Members of an object of aggregate or union type share the storage instance with their defining object. Objects that do not originate from definitions and that are not explicitly created within a storage instance by means of effective type, such as compound literals, string literals, or temporary objects may share or reuse storage instances in unspecified ways, provided that the lifetime of the object is included in the lifetime of the storage instance.[40]

4 ~~An~~ The storage instance of an object whose identifier is declared without the storage-class specifier ~~**`_Thread_local`**,~~ **`thread_local`**, and either with external or internal linkage or with the storage-class specifier ~~**`static`**,~~ **`static`**, has *static storage duration* ~~. Its~~ , as do storage instances for string literals, and some compound literals and lambda values. The lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

5 ~~An~~ The storage instance of an object whose identifier is declared with the storage-class specifier ~~**`_Thread_local`**~~ **`thread_local`** has *thread storage duration*. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There

[37] There is only one name space for tags even though three are possible.

[38] The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

[39] In the case of a volatile object, the last store need not be explicit in the program.

[40] In particular, such an object need not have a unique address, and, if suitable for their concrete value, string literals, compound literals or certain objects with temporary lifetime may overlap.

is a distinct ~~object~~ instance of the object and associated storage per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 ~~An~~ The storage instance of an object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*, as ~~do~~ are storage instances of temporary objects and some compound literals ~~.~~ or lambda values. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

7 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object and associated storage is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

8 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[41] If the scope is entered recursively, a new instance of the object and associated storage is created each time. The initial value of the object is indeterminate.

9 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to ~~an object~~ a *temporary object* with automatic storage duration and *temporary lifetime*.[42] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime ~~results in~~ has undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. ~~Such an object need not have a unique address.~~

10 **NOTE** C and C++ diverge on their concepts for auxiliary objects. In particular in C++ there is no concept that would be similar to compound literals in C, namely of a temporary unnamed object that has a lifetime of the surrounding scope. In C++, all temporary objects are more similar to objects of temporary storage duration as they are defined above, only that references to them may be taken without restriction on the type.

If addresses of compound literals are taken and passed into functions, they may leak to places in the program that are difficult to foresee. To be portable in the C/C++ core, application code should always ensure that addresses of compound literals are not used in a wider range than within the expression in which they are defined.

Implementations are invited, as much as this is possible, to diagnose the usage of compound literals outside of their originating expression.

Forward references: object definitions (6.7), aggregate or union type (6.2.5), array declarators (6.7.8.2), compound literals (6.5.2.5), declarators (6.7.8), function calls (6.5.2.2), initialization (6.7.12), statements (6.8), effective type (6.5).

6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).[43] Additionally, there are

[41] Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

[42] The address of such an object is taken implicitly when an array member is accessed.

[43] A type can be incomplete or complete throughout an entire translation unit, or it can change states at different points within a translation unit.

opaque object types that are types that have internal state but no accessible value.[44)]

2 An object declared as type ~~`_Bool`~~ `bool` is large enough to store the values ~~0 and 1~~`false` and `true`.

3 An object declared as type **`char`** is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a **`char`** object, its value is guaranteed to be nonnegative. If any other character is stored in a **`char`** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

4 There are five *standard signed integer types*, designated as **`signed char`**, **`short int`**, **`int`**, **`long int`**, and **`long long int`**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.[45)] The standard and extended signed integer types are collectively called *signed integer types*.[46)]

5 An object declared as type **`signed char`** occupies the same amount of storage as a "plain" **`char`** object. A "plain" **`int`** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **`INT_MIN`** to **`INT_MAX`** as defined in the header `<limits.h>`).

6 For each of the signed integer types, there is a *corresponding* (but different) *unsigned integer type* (designated with the keyword **`unsigned`**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type ~~`_Bool`~~ `bool` and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard and extended unsigned integer types are collectively called *unsigned integer types*.[47)]

7 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

8 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[48)] A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

10 There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**.[49)] The set of values of the type **`float`** is a subset of the set of values of the type **`double`**; the set of values of the type **`double`** is a subset of the set of values of the type **`long double`**.

11 There are three *complex types*, designated as ~~`float _Complex`,~~ `complex_type(float)`, ~~`double _Complex`,~~ `complex_type(double)`, and ~~`long double _Complex`~~ `complex_type(long double)`. ~~(Complex types are a conditional feature that implementations need not support; see 6.10.8.2.)~~ The real floating and complex types are collectively called the *floating types*.

12 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the ~~type given by deleting the keyword~~

44) Opaque types defined by this specification are **`atomic_flag`**, **`cnd_t`**, **`fenv_t`**, **`fexcept_t`**, **`FILE`**, **`jmp_buf`**, **`mtx_t`**, **`once_flag`**, **`va_list`**, and **`void`**, which are complete types, and aggregate or union types that are entirely composed of such types. Opaque types can be complete, such that objects of such a type can be defined and initialized, and such that the **`decltype`**, **`sizeof`**, **`alignof`** and address-of operators can be applied to them, but they are such that no other operation such as evaluation or assignment is defined for them. In particular, opaque types can neither be copied by assignment, nor, unless specified otherwise, by **`memcpy`** or byte-wise copy.

45) Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

46) Therefore, any statement in this document about signed integer types also applies to the extended signed integer types.

47) Therefore, any statement in this document about unsigned integer types also applies to the extended unsigned integer types.

48) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

49) See "future language directions" (6.11.1).

~~**_Complex** from the type name~~floating type from which it is derived when using the **complex_type** macro as above.

13 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

14 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[50)]

15 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[51)]

16 An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

17 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. The integer and real floating types are collectively called *real types*.

18 Integer and floating types are collectively called *arithmetic types*.[52)] Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

19 The **void** type comprises an empty set of values; it is ~~an incomplete object type that cannot be completed~~a complete opaque object type with a size of 1.

20 Any number of *derived types* can be constructed from the object and function types, as follows:

- An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". The construction of an array type from an element type is called "array type derivation".

- A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

- A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

- A *lambda type* is a complete object type that describes the value of a lambda expression. A lambda type is characterized but not determined by a return type that is inferred from the function body of the lambda expression, and by the number, order, and type of parameters that are expected for function calls.

- A *function type* describes a function with specified return type. A function type is characterized by its return type~~and the number~~, the number, order and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". The construction of a function type from a return type is called "function type derivation".

- A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced

[50)]An implementation can define new keywords that provide alternative ways to designate a basic (or any other) type; this does not violate the requirement that all basic types be different. Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

[51)]**CHAR_MIN**, defined in `<limits.h>`, will have one of the values 0 or **SCHAR_MIN**, and this can be used to distinguish the two options. Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either.

[52)]Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

type. If the type is an object type, the pointer also carries a *provenance*, typically identifying the storage instance holding the corresponding object, if any. A pointer value is *valid* if and only if it has a non-empty provenance, there is a live storage instance for that provenance, and the address is either within or one-past the addresses of that storage instance. It is *null* to indicate that it does not refer to such a function or object,[53] and *indeterminate* otherwise. A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.[54] Under certain circumstances a pointer value can have an address that is the end address of one storage instance and the start address of another. It (and any pointer value derived from it by means of arithmetic operations) shall then only be used with one and the same of these provenances as operand to subsequent operations that require a provenance.

— An *atomic type* describes the type designated by the construct ~~**_Atomic**(*type-name*). (Atomic types are a conditional feature that implementations~~ **atomic_type**(*type-name*). ~~need not support; see 6.10.8.2.)~~ The representation, size and set of admissible values of an atomic type is the same as the type from which it is derived, but alignment requirements may be more strict.

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[55]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope. An aggregate or union type is opaque, if all of its members are opaque.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[56] corresponding to the combinations of one ~~, two, or all three~~ or two of the **const** ~~, **volatile**, and **restrict**~~ and **volatile** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[57] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 ~~Further, there is the **_Atomic** qualifier. The presence of the **_Atomic** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic,~~

[53] A pointer object can be null by implicit or explicit initialization or assignment with a null pointer constant or by another null pointer value. A pointer value can be null if it is either a null pointer constant or the result of an lvalue conversion of a null pointer object. A null pointer will not appear as the result of an arithmetic operation.

[54] The provenance of a pointer value and the property that such a pointer value is indeterminate are generally not observable. In particular, in the course of the same program execution the same pointer representation (6.2.6) may refer to objects with different provenance and may sometimes be valid and sometimes be indeterminate. Yet, this information is part of the abstract state machine and may restrict the set of operations that can be performed on the pointer.

[55] Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[56] See 6.7.3 regarding qualified array and function types.

[57] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

~~qualified, or unqualified type" whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.~~

A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[57] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. ~~Pointers to other types need not~~ It is implementation-defined if other groups of pointer types have the same representation or alignment requirements.[58]

28 **NOTE** Neither C nor C++ currently have the explicit concept of opaque types. It is introduced here, such that this core specification may better accommodate C with the implicit initialization properties that C++ provides for types that are not copyable.

29 **EXAMPLE 1** The type designated as "`float *`" has type "pointer to `float`". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type — its type is "pointer to const-qualified `float`" and is a pointer to a qualified type.

30 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float`. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7), predefined macros (6.10.8).

6.2.5.1 Predefined types

1 The following types shall be defined with the indicated names:

type	name	integer category
`decltype(nullptr)`	`nullptr_t`	none
`decltype(((char*)0)-((char*)0))`	`ptrdiff_t`	signed
`decltype(sizeof 1)`	`size_t`	unsigned
`decltype(+L""[0])`	`wchar_t`	signed or unsigned
`decltype(+u8""[0])`	`char8_t`	character
`decltype(+u""[0])`	`char16_t`	unsigned
`decltype(+U""[0])`	`char32_t`	unsigned

It is implementation-defined if any of these, other than **`nullptr_t`**, represent proper types or are provided as an alias as if by **`typedef`** to one of the basic integer types as previously defined. If any such type is a proper type, the value of the corresponding feature test macro in 6.10.8.1 expands to **`true`** and the type is added to the type categories (and all categories that include any of these) as specified. If the category is signed, there shall be a corresponding unsigned integer type that is also a proper type; if the category is unsigned, there shall be a corresponding signed integer type that is also a proper type. If such a type has a width less than or equal to **`int`**, it has a conversion rank lower than **`int`**. Otherwise it has a conversion rank that is different from any other integer type, and, if it has a width that is less than or equal to the width of an integer type T other than one of those defined in this subclause as a proper type, its conversion rank is less than or equal to the rank of T.[59]

2 These types shall have the following properties:

— **`nullptr_t`** is the type of the **`nullptr`** constant. This is an unspecified type that has the same size as a character pointer.

— **`ptrdiff_t`** is a signed integer type that is the result of subtracting two pointers.

— **`size_t`** is the unsigned integer type that is the result of the **`sizeof`** operator, the **`alignof`** operator and the **`offsetof`** macro.

[58] An implementation might represent all pointers the same and with the same alignment requirements.

[59] These rules are chosen, such that promotion and arithmetic conversion has not to take special considerations for these types.

- **wchar_t** is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. Each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant. It is implementation-defined if **wchar_t** is any of the basic integer types as defined above or if it is distinct from these.
- **char8_t** is an integer type used to encode the bytes of UTF–8 multi-byte characters as defined by ISO/IEC 10646.[60] It has the same alignment and representation as a character type.
- **char16_t** is an unsigned integer type used for UTF–16 encoded characters as defined by ISO/IEC 10646. It has the same alignment and representation as **uint_least16_t** (described in 7.20.1.1).
- **char32_t** is an unsigned integer type used for UTF–32 encoded characters as defined by ISO/IEC 10646. It has the same alignment and representation as **uint_least32_t** (also described in 7.20.1.1).

3 Additionally there is **max_align_t**, which is an object type whose alignment is the greatest fundamental alignment.

4 **NOTE** For C, these identifiers are usually **typedef**, traditionally provided through C library headers. In particular **char8_t** is one of the character types, and **char16_t** and **char32_t** have the same type as **uint_least16_t** and **uint_least32_t**, respectively. C++ has **nullptr_t**, **wchar_t**, **char8_t**, **char16_t** and **char32_t** as proper types that are distinct from any other basic type.

Recommended practice

5 The types used for **size_t** and **ptrdiff_t** should not have an integer conversion rank greater than that of **signed long int** unless the implementation supports objects large enough to make this necessary.

Forward references: Predefined identifiers (6.4.2.2), additive operators (6.5.6), the **sizeof** and **alignof** operators (6.5.3.4), alignment (6.2.8), expression types 6.7.11, localization (7.11), mandatory type and value macros .

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in 6.2.5 and in this subclause. An object is represented (or held) by a storage instance (or part thereof) that is either created by an allocation (for allocated storage duration), at program startup (for static storage duration), at thread startup (for thread storage duration), or when the lifetime of the object starts (for automatic storage duration).

2 ~~Except for bit-fields, objects~~ An addressable storage instance[61] of size m provides access to a byte array of length m. All bytes of the array have an *abstract address* , which is a non-negative integer value that is determined in an implementation-defined manner. The abstract addresses of the bytes are increasing with the ordering within the array, and they shall be unique and constant during the lifetime. The address of the first byte of the array is the *start address* of the storage instance, the address one element beyond the array at index m is its *end address* . The abstract addresses of the bytes of all storage instances of a program execution form its *address space* . A storage instance Y *follows* storage instance X if the start address of Y is greater or equal than the end address of X, and it *follows immediatly* if they are equal. During the common lifetime of any two distinct addressable storage instances X and Y, either Y follows X or X follows Y in the address space. This document imposes no other constraints about such relative position of addressable storage instances whenever they are created.[62]

[60] This effectively means that such characters that have a one-byte UTF–8 encoding are encoded using an ASCII encoding.

[61] All storage instances that do not originate from an object definition with **core :: noalias** attribute are addressable by using the pointer value that was returned by their allocation (for allocated storage duration) or by applying the address-of operator & (6.5.3.2) to the object that gave rise to their definition (for other storage durations).

[62] This means that no relative ordering between storage instances and the objects they represent can be deduced from

3 Unless stated otherwise, a storage instance is *exposed* if a pointer value p of effective type T* with this provenance is used in the following contexts:[63)]

— Any byte of the object representation of p is used in an expression.[64)]

— Any byte of the object representation of p is passed to the **fwrite** library function.

— p is converted to an integer.

— p is used as an argument to a %p conversion specifier of the **printf** family of library functions.

— p is used as an argument to the **totext** type-generic macro or any of the related features such that a textual conversion of the pointer value is stored or written ot an output stream.

Other provisions of this document not withstanding, if the object representation of p is read through an lvalue of a pointer type S* that has the same representation and alignment requirements as T*, that lvalue has the same provenance as p and the provenance is not exposed.[65)] Exposure of a storage instance is irreversible and constitutes a side effect in the abstract state machine.

4 Unless stated otherwise, pointer value p is *synthesized* if it is constructed by one of the following:[66)]

— Any byte of the object representation of p is changed

- by an explicit byte operation,
- by type punning with a non-pointer object or with a pointer object that only partially overlaps,
- or by a call to **memcpy** or similar function that does not write the entire pointer representation or where the source object does not have an effective pointer type.

— Any byte of the object representation of p is passed to the **fread** library function.

— p is converted from an integer value.

— p is used as an argument to a %p conversion specifier of the **scanf** family of library functions.

Special provisions in the respective clauses clarify when such a synthesized pointer is a null, valid, or indeterminate.

5 Objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

6 ~~Values stored in unsigned bit-fields and objects~~ Objects of type **unsigned char** shall be represented using a pure binary notation.[67)]

7 Values stored in ~~non-bit-field~~ objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. ~~The value may be copied into an object of type~~ Converting

syntactic properties of the program (such as declaration order or order inside a parameter list) or sequencing properties of the execution (such as one instantiation happening before another).

63) Pointer values with exposed provenance may alias in ways that cannot be predicted by simple data flow analysis.

64) The exposure of bytes of the object representation can happen through a conversion of the address of a pointer object containing p to a character type and a subsequent access to the bytes, or by storing p in a **union** that allows access to all or parts of the object representation by means of a type that is not a pointer type or by a pointer type that gives rise to a different object representation.

65) This means that pointer members in a **union** can be used to reinterpret representations of different character and void pointers, different **struct** pointers, different **union** pointers or pointers with differently qualified target types.

66) Pointer values with synthesized provenance may alias in ways that cannot be predicted by simple data flow analysis.

67) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

a pointer of such an object to a pointer to a character type or **void** yields a pointer into the byte array of the storage instance such that the values of the first n ~~] (e.g., by **memcpy**); the resulting set of bytes is called the of the value. Values stored in bit-fields consist of m bits, where m is~~ bytes determine the value of the object; the position of the first byte of these in the byte array is the ~~size specified for the bit-field~~*byte offset* of the object in its storage instance, the converted address is called the *byte address* of the object, and the set of bytes is called the *object representation* of the value. The object representation ~~is the set of m bits the bit-field comprises in the addressable storage unit holding it.~~ may be used to copy the value of the object into another object (e.g., by **memcpy**). Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations. The object representations of pointers and how they relate to the abstract addresses they represent are not further specified by this document.

8 Certain object representations need not represent a value of the object type. If ~~the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If~~ such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[68] Such a representation is called a trap representation.

9 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values and bytes that correspond to opaque members have an indeterminate state.[69] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

10 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

11 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[70] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

12 ~~Loads and stores of objects with atomic types~~ All operations on atomic objects ~~are done with **memory_order_seq_cst** semantics.~~ that do not specify otherwise have **memory_order_seq_cst** memory consistency. If an operation with identical values on the non-atomic type is erroneous,[71] the atomic operation results in an unspecific object representation, that may or may not be an invalid value for the type, such as an invalid address or a floating point NaN. Thereby such an operation may by itself never raise a signal, a trap, or result otherwise in an interruption of the control flow.[72]

Forward references: declarations (6.7), expressions (6.5), address and indirection operators (6.5.3.2), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3), input/output (7.21).

6.2.6.2 Integer types

1 For unsigned integer types the bits of the object representation shall be divided into two groups: value bits and padding bits. If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.The number of value bits N is called the *width* of the

[68]Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[69]Thus, for example, structure assignment need not copy any padding bits or members that have an opaque type.

[70]It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. In particular, if $\equiv$ is defined for type `T`, then `x` $\equiv$ `y` does not imply that `memcmp(\&x, \&y, sizeof (T))` $\equiv$ `0`. Furthermore, `x` $\equiv$ `y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

[71]Such erroneous operations may for example incur arithmetic overflow, division by zero or negative shifts.

[72]Whether or not an atomic operation may be interrupted by a signal depends on the lock-free property of the underlying type.

unsigned integer type. There need not be any padding bits; **`unsigned char`** shall not have any padding bits.

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. If the corresponding unsigned type has width N, the signed type uses the same number of N bits, its *width*, as value bits and sign bit. $N-1$ are value bits and the remaining bit is the sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type. If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, it has value $-(2^{N-1})$. There need not be any padding bits; **`signed char`** shall not have any padding bits.

3 The values of any padding bits are unspecified. A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

4 The *precision* of an integer type is the number of value bits.

5 **NOTE 1** Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

6 **NOTE 2** The sign representation defined in this document is called *two's complement*. Previous revisions of this document additionally allowed other sign representations.

7 **NOTE 3** For unsigned integer types the width and precision are the same, while for signed integer types the width is one greater than the precision.

6.2.7 Compatible type and composite type

1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.8 for declarators.[73] Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For two structures, corresponding members shall be declared in the same order ~~. For two structures or unions, corresponding bit-fields shall~~ and have the same ~~widths~~**`core::noalias`** and **`core::alias`** attributes. For two enumerations, corresponding members shall have the same values.

2 All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

3 A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

— If both types are array types, the following rules are applied:

- If one type is an array of known constant size, the composite type is an array of that size.
- Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
- Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
- Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
- Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

[73] Two types need not be identical to be compatible.

The element type of the composite type is the composite type of the two element types.

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.

— If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[74] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

Forward references: array declarators (6.7.8.2).

5 **EXAMPLE** Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the ~~`_Alignas`~~ `alignas` keyword.

2 A *fundamental alignment* is a valid alignment less than or equal to ~~`_Alignof (max_align_t)`~~ `alignof(max_align_t)`. Fundamental alignments shall be supported by the implementation for objects of all storage durations. The alignment requirements of the following types shall be fundamental alignments:

— all atomic, qualified, or unqualified basic types;

— all atomic, qualified, or unqualified enumerated types;

— all atomic, qualified, or unqualified pointer types;

— all array types whose element type has a fundamental alignment requirement;

— all types specified in Clause 7 as complete object types;

— all structure or union types all of whose elements have types with fundamental alignment requirements and none of whose elements have an alignment specifier specifying an alignment that is not a fundamental alignment.

3 An *extended alignment* is represented by an alignment greater than ~~`_Alignof (max_align_t)`~~ `alignof (max_align_t)`. It is implementation-defined whether any extended alignments are supported and the storage durations for which they are supported. A type having an extended alignment requirement is an *over-aligned* type.[75]

4 Alignments are represented as values of the type `size_t`. Valid alignments include only fundamental alignments, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.

[74] As specified in 6.2.1, the later declaration might hide the prior declaration.

[75] Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an ~~`_Alignof`~~ `alignof` expression. The types **`char`**, **`signed char`**, and **`unsigned char`** shall have the weakest alignment requirement.

7 Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

6.2.9 Mutual representability of types and objects

1 For the purpose of interoperability between functions and their callers, in the above clauses and in some library clauses, several representations of types are required to be the same. Type representability extends this concept to aggregate and union types. When restricted to types such that no members or elements are pointers, flexible array members, atomics, or opaque types the concept of representability models a situation that is similar to type punning through unions. The type of union member A can be represented by the type of a wider union member B, if for any valid representation for the type of B, union member A can be modified in any permitted way and the result remains a valid representation for B.

2 **NOTE** This document requires that the following groups of types have the same representation:

— Qualified versions of the same type.

— Integer types with same width that have no padding bits.[76]

— Complex types and two element vectors of the corresponding real type.

— Pointers to character types and **`void`**.

— Pointers to structure types.

— Pointers to union types.

— The atomic integer types (7.17.6) and their corresponding direct type.

Other types may form such groups or some of the above groups might fuse to larger groups with the same representation in an implementation-defined way. For example many implementations with a flat address space represent all pointers (data and function) the same. For types that contain no pointers, such implementation-defined properties of representations should only have an impact for the following definitions by the sizes of integer types, and by the fact of whether or not these have padding. On the other hand, definitions for types that contain pointer types to non-character types as elements or members are strongly affected by implementation specific choices concerning representations of pointers.

3 A type is *primitive* if it is not an aggregate or union type. If T is an aggregate or union type, a direct leaf is a *pack* or an element or member e, if it is primitive or a flexible array member; a *leaf* of T is a direct leaf of T or recursively a leaf of one of its elements or members. The offset $o(T, e)$ of a leaf e in T is is the byte offset of e in an object of type T.[77]

4 An *interval* I of a type τ is a construct to describe sequences of adjacent elements that have all the same representation as τ such that an array of type $\tau[k]$ can stand in for the whole sequence:

— For T an aggregate or union type, an *interval* I of an arithmetic type τ is a sequence of k leafs all having the same representation as τ and such that they follow each other in the representation of T without padding.[78] The *interval array type* $A(I)$ is $\tau[k]$, the size of the interval is **`sizeof`**$(A(I))$ and the offset $o(T, I)$ of the interval is the offset of its first element.

— A *pack* e may only occur as the single element of an interval I said to have the size s and offset of the *pack* and the interval array type $A(I)$ then are both $void$`[s]`.

[76] In particular, **`char`**, **`signed char`** and **`unsigned char`** have the same representation.

[77] Note that a union type in general has several leaves at offset 0, and so types that recursively contain union types may have several leaves at a particular offset.

[78] That is $o(T, e_{i+1})$ is $o(T, e_i) +$ **`sizeof`**(t) for all i.

— A flexible array member e of type `s[]` may only occur as the single element of an interval I said to be of infinite size and the type of the interval and the interval array type $A(I)$ then are both `s[]`.

— Any other member e with type τ of an aggregate or union type forms an interval $I(e)$ of its own, with $A(I)$ being of type $\tau[1]$ and offset $o(T, e)$.

— If T itself is a primitive type it is said to be its own leaf, it has exactly one interval, (T), with interval array $T[1]$ and offset 0.

An *interval partition* $\mathcal{I}$ of T is a partition of the set of leaves of T into intervals.[79)]

5 We say that an interval partition $\mathcal{I}$ of one type embeds in a partition $\mathcal{J}$ of another type if intervals of $\mathcal{I}$ can be mapped on intervals of $\mathcal{J}$ by respecting offset and size, that is if $\mathcal{I}$ has the same decomposition into "arrays" of storage units. More precisely, let T and S be two types and $\mathcal{I}$ and $\mathcal{J}$ be interval partitions for T and S, respectively. A mapping $f : \mathcal{I} \to \mathcal{J}$ is an *embedding* of T into S if for all $I \in \mathcal{I}$, I and $f(I)$ are intervals of the same offset and size,[80)] and for all $J \in \mathcal{J}$ with $o(S, J) <$ **sizeof**(T) there is $I \in \mathcal{I}$ with $J = f(I)$.

6 To extend embeddability to representations, we will use recursion on the structure of the types. For the bottom of this recursion we first consider array types, $t[N]$ and $s[M]$, that have the same size (**sizeof**$(t[N]) \equiv$ **sizeof**$(s[M])$) and such that t and s are primitive types. $t[N]$ is said to be representable by $s[M]$ if s has no more qualifiers than t and if one of the following holds:

— t and s are compatible types.

— t and s are basic types and have the same representation.

— t and s are pointers to character type or **void**, $u*$ and $v*$, respectively, such that v has no more qualifiers than u.

— One of t or s is a complex type and the other is the corresponding real type.

— Both t and s are integer types without padding.

If both have the same qualifiers, the array types are said to be *mutually representable*.[81)]

7 To extend this notion recursively to be arbitrary object types T and S, we assume that S does not have more qualifiers than T, and, if both are complete types, **sizeof**$(T) \leq$ **sizeof**(S). Then T is *representable* by S if one of the following holds:

— T and S are array types of base t and s, respectively, S is incomplete and t`[`**sizeof**(s)`]` can be represented by s`[`**sizeof**(t)`]`.

— T has no flexible array members, S is a structure or union type with flexible array members, and there is an integer n such that T is representable by the type S' where each flexible array member e of type `s[]` is replaced with an array e' of type s`[`n`]` at the same offset.

— If T and S are pointer types $t*$ and $s*$, respectively, that have the same representation, such that neither is a pointer to character type or **void**:

  - t and s are both structure or union types with a flexible array member such that t is representable by s.
  - t`[]` and s`[]` are mutually representable.

[79)] If the type T is a union type or contains a union type, byte ranges within the representation of different intervals may overlap.

[80)] This means in particular that intervals for flexible array members can only map or be mapped by intervals with the same property.

[81)] For the first three cases, N and M must be equal, and for the complex and real case one must be twice the other.

— There are $\mathcal{I}$ and $\mathcal{J}$ interval partitions for T and S, respectively, and an embedding $f : \mathcal{I} \to \mathcal{J}$ such that for all $I \in \mathcal{I}$, $A(I)$ is representable by $A(F(I))$, or additionally, if I is (e) for a flexible array member e, $A(F(I))$ is an incomplete array of character type that is not more qualified than e.

T and S are *mutually representable* if in addition they have the same qualifications and the same size. Two union or structure packs e and f are said to be mutually representable, if the union or structure types that have their respective sequences as members are mutually representable.

8 The *effective size* of an object represented by an lvalue A of type T_A is the size that it occupies within its provenance; if T_A is a complete type that has no flexible array members it is **sizeof**(T_A); if T_A has flexible array members or is an incomplete array type it is the number of bytes from the first byte of A to the end of the provenance. An lvalue A of type T_A is representable by an object B of type T_B if

— the alignment of B is a valid alignment for type T_A,

— the effective size of A is less than or equal to the effective size of B, and

— either

- T_A is representable by T_B, and for each leaf e' of pointer type $s*$ in T_B with $o(T_B, f) <$ **sizeof**(T_A) and the corresponding pointer leaf e of type $t*$ in T_A, such that e' in B has a valid non-null value p, the lvalue $*(t*)p$ is representable by $*p$, or
- B is a character array that is not more qualified than T_A.

9 **EXAMPLE 1** For the following types we assume that there is no padding between or after the elements of `A`, `B`, `C`, and `D`:

```
typedef struct { double d; } A;
typedef struct { A a; double e; int i; } B;
typedef union  { B b; int j; } C;
typedef struct { double t[3]; } D;
typedef struct { size_t len; double const ddat[]; } dvec;
typedef struct { size_t len; complex_type(double) cdat[]; } cvec;
typedef struct { size_t len; alignas(double) unsigned char bdat[]; } bvec;
```

The following table shows examples where T is representable by S or not:

T	S	repres.	
`signed`	`unsigned`	yes	
`unsigned`	`signed`	yes	
`double`	`complex_type(double)`	yes	maps to the real part
`complex_type(double)`	`double[2]`	yes	
`complex_type(double)[k]`	`double[2*k]`	yes	
`double[2*k]`	`complex_type(double)[k]`	yes	
`complex_type(double)[]`	`double[]`	yes	
`double[]`	`complex_type(double)[]`	yes	
`dvec`	`bvec`	yes	
`bvec`	`dvec`	no	base type of `bdat` does not fit
`dvec`	`cvec`	yes	
`cvec`	`dvec`	no	member `ddat` is **const** qualified
`complex_type(double)`	`D`	yes	
`complex_type(double)[3]`	`D[2]`	yes	
`complex_type(double)[sizeof(D)]`	`D[sizeof(complex_type(double))]`	yes	
`complex_type(double)[]`	`D[]`	yes	
`D`	`complex_type(double)`	no	too small
`D[2]`	`complex_type(double)[3]`	yes	
`D[sizeof(complex_type(double))]`	`complex_type(double)[sizeof(D)]`	yes	
`D[]`	`complex_type(double)[]`	yes	

`double`	`A`	yes	via member `d`
`A[42]`	`A[]`	yes	
`A[]`	`A[42]`	no	incomplete array needs another incomplete array
`double`	`B`	yes	via member `a`
`complex_type(double)`	`B`	yes	via members `a.d` and `e`
`A`	`B`	yes	same
`A[1]`	`B`	yes	same
`A[2]`	`B[1]`	yes	via element `[0]`
`A[3]`	`B[2]`	no	element [2] is not representable
`int`	`C`	no	no `int` leaf at offset 0
`B`	`C`	yes	via member `b`
`A`	`C`	yes	recursively via member `b.a`
`double`	`C`	yes	recursively via member `b.a.d`

6.3 Conversions

1 Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.

2 Unless explicitly stated otherwise, conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (6.5.4).

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

1 Every integer type has an *integer conversion rank* defined as follows:

— No two signed integer types shall have the same rank, even if they have the same representation.

— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

— The rank of **`long long int`** shall be greater than the rank of **`long int`**, which shall be greater than the rank of **`int`**, which shall be greater than the rank of **`short int`**, which shall be greater than the rank of **`signed char`**.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

— The rank of **`char`** shall equal the rank of **`signed char`** and **`unsigned char`**.

— The rank of **`bool`** shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

2 The following may be used in an expression wherever an **`int`** or **`unsigned int`** may be used:

— An object or expression with an integer type (other than **`int`** or **`unsigned int`**) whose integer conversion rank is less than or equal to the rank of **`int`** and **`unsigned int`**. ~~A bit-field of type **`_Bool`**, **`int`**, **`signed int`**, or **`unsigned int`**.~~

If an **`int`** can represent all values of the original type~~(as restricted by the width, for a bit-field)~~, the value is converted to an **`int`**; otherwise, it is converted to an **`unsigned int`**. These are called the *integer promotions*.[82] All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a "plain" **`char`** can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

6.3.1.2 Boolean type

1 When any scalar value is converted to **`bool`**, the result is ~~0~~ **`false`** if the value ~~compares equal to~~ is 0; otherwise, the result is ~~1~~ **`true`**.[83]

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than ~~**`_Bool`**~~ **`bool`**, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[84]

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6.3.1.4 Real floating and integer

1 When a finite value of real floating type is converted to an integer type other than ~~**`_Bool`**~~ **`bool`**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[85]

2 When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

6.3.1.5 Real floating types

1 When a value of real floating type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is

[82] The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary `+`, `-`, and `~` operators, and to both operands of the shift operators, as specified by their respective subclauses.

[83] NaNs do not compare equal to 0 and thus convert to **`true`**.

[84] The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

[85] The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. Thus, the range of portable real floating values is $(-1, \texttt{U}\mathit{type}\texttt{\_MAX} + 1)$.

undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

6.3.1.6 Complex types

1 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

6.3.1.7 Real and complex

1 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

~~When a value of complex type is converted~~

2 **NOTE** C and C++ differ much in their strategies for conversions of complex types, in particular for conversions from complex types to real types. Therefore, this specification only defines a conversion from a complex type to a real type ~~other than `_Bool`,~~[86] if that type is **`bool`**, see 6.3.1.2. C is more permissive, and allows more conversions to real types that just drop the imaginary part of ~~the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.~~ a complex number. Applications that target the common C and C++ core should not use that feature but use an explicit call to **`real_value`** instead.

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

> First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**.
>
> Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**.
>
> Otherwise, if the corresponding real type of either operand is **`float`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`float`**.[86]
>
> Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
>> If both operands have the same type, then no further conversion is needed.
>>
>> Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
>>
>> Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
>>
>> Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
>>
>> Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

[86] ~~See 6.3.1.2.~~

[86] For example, addition of a **`complex_type`**`(`**`double`**`)` and a **`float`** entails just the conversion of the **`float`** operand to **`double`** (and yields a **`complex_type`**`(`**`double`**`)` result).

2 The values of floating operands and of the results of floating expressions may be represented in greater range and precision than that required by the type; the types are not changed thereby. See 5.2.4.2.2 regarding evaluation formats.

3 **NOTE** C and C++ differ much in their strategies for arithmetic of complex types, in particular for operations that mix complex types and real types, which are generally not defined for C++. Implementations that target the common C and C++ core must provide means to circumvent problems that may originate in that restriction of C++. For example, they may offer overloaded wrappers for all four arithmetic operations that are defined for complex numbers, such that the other operand can be any real type.

6.3.2 Other operands

Constraints

1 No evaluation shall be formed that has a result that is an object with

— an incomplete type;

— an opaque type, other than a as void expression, or as the operand of a cast to **`void`**.

2 No evaluation shall be formed that has an operand that is an object with

— an incomplete type that is not an array, other than for the unary & operator;

— an opaque type, other than for the the **`sizeof`** operator, the **`alignof`** operator, or the unary & operator, or, if it is an opaque array type, for array to pointer conversion.

6.3.2.1 Lvalues, arrays, function designators and lambdas

1 An *lvalue* is an expression ~~(with an object type other than **void**)~~ that potentially designates an object;[87] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have an opaque type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.[88]

2 For an lvalue expression that has not an enumerated, array or function type the *generic type* is the non-atomic type of the lvalue with all qualifiers dropped; for an expression that has an enumerated type, it is the non-atomic compatible integer type with all qualifiers dropped; for an expression that has type "array of *type*" it is "pointer to *type*"; for a function designator with type "function with prototype *type*" it has type "pointer to function with prototype *type*". For any other expression it is the type of the expression. Unless specified otherwise in the following, the evaluation of an lvalue expression yields a value with the generic type of the lvalue.

3 Except when it is the operand of the **`decltype`** specifier, the **`sizeof`** operator, the **`alignof`** operator, the unary & operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. ~~If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the~~ The behavior is undefined if one of the following conditions hold:

— The lvalue does not designate an object when it is evaluated.

[87] The name "lvalue" comes originally from the assignment expression `E1 = E2`, in which the left operand `E1` is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this document described as the "value of an expression".

An obvious example of an lvalue is an identifier of an object. As a further example, if `E` is a unary expression that is a pointer to an object, `*E` is an lvalue that designates the object to which `E` points.

[88] This means in particular that a structure or union type that contains some members that are opaque and some that are not can still be the type of a modifiable lvalue.

— The object representation is a trap representation for the type.[89)]

— The lvalue designates an object of automatic storage duration that ~~could have been declared with the **register** storage class (never had its address taken), and that object is uninitialized (not declared~~ was not defined with an initializer~~and~~ , no assignment to it has been performed prior to use~~), the~~ , and the unary & operator and array-to-pointer conversion are never applied to the object.[90)]

4 Additionally, if the type is a pointer type `T*`, a pointer value and an associated provenance, if any, is determined as follows:

— If the object representation represents a null pointer the result is a null pointer.

— If the last store to the representation array was with a pointer type `S*` that has the same representation and alignment requirements as `T*`, the result is the same address and provenance as the stored value.

— Otherwise, the object representation of the lvalue shall represent an abstract address within (or one-past) an exposed storage instance, such that the exposure happened before this lvalue conversion, and the result has that address and provenance.[91)]

The behavior is undefined if the lvalue conversion does not happen during the lifetime of the associated provenance, the address is not a valid address (or one-past) for the associated provenance, or the address is not correctly aligned for the type.

5 Except when it is the operand of the **decltype** specifier, the unary **sizeof** operator, or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. ~~If the array object has register storage class, the behavior is undefined.~~

6 A *function designator* is an expression that has function type. Except when it is the operand of the **decltype** specifier, the**sizeof** operator,[92)] or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

7 Lambda types originating from lambda expressions with captures shall not be converted to any other object type. A lambda value originating from a function literal with a type "lambda with prototype *type*" can be converted implicitly or explicitly to an expression that has type "pointer to function with prototype *type*".[93)]

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.17), ~~common definitions (7.19~~expression types (6.7.11), initialization (6.7.12), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and ~~**_Alignof**~~ **alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 `void`

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

89) Character types have no trap representation, thus reading representation bytes of an addressable live storage instance is always defined.

90) The lvalue is necessarily the result of the evaluation of an identifier. This requirement is not a constraint, because complicated control flows might make the detection of such an error difficult. It is recommended that implementations diagnose such situations as good as they may. If detected, it does not suggest that the corresponding execution path is unreachable, but that a programming error has occurred.

91) If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

92) Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraints in 6.5.3.4.

93) Since lambdas of different type cannot be assigned to each other, in the conversion of a function literal to a function pointer, the prototype of the originating lambda expression can be assumed to be known and a diagnostic can be issued if the prototypes do not aggree.

6.3.2.3 Pointers

1 A pointer to **`void`** may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to **`void`** and back again; the result shall compare equal to the original pointer.

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, ~~or~~ such an expression cast to type **`void`** `*`, ~~is called~~ or the constant **`nullptr`** are all a *null pointer constant*.[94] If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function. If the constant **`nullptr`** is converted to a type other than a pointer type or **`bool`**, the behavior is undefined.

4 Conversion of a null pointer to ~~another~~ a pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. If the source type is signed, the operand is first converted to the corresponding unsigned type. The result is then determined in the following order:

- The operand has a value that could have been the result of the conversion of a null pointer value. The result is a null pointer.
- The operand is an abstract address within or one past a live and exposed storage instance, such that the exposure happened before this integer-to-pointer conversion. The conversion synthesizes a pointer value with that address, provenance and target type.[95]
- The pointer value is indeterminate.

Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

6 A pointer value may be converted to **`bool`**.[96] The result is **`false`** if it is a null pointer and **`true`** if it is valid. Otherwise the behavior is undefined.

7 ~~Any~~ Otherwise, any pointer type may be converted to an integer type. ~~Except as previously specified~~For a null pointer, the result is chosen from a non-empty set of implementation-defined ~~. If the result cannot be represented in the integer~~ values.[97] If the pointer value is valid, its provenance is henceforth exposed. Except as previously specified, the result is the bit representation of the abstract address interpreted in the target type. If the abstract address has more significant bits than the width of the target type, the behavior is undefined. The result need not be in the range of values of any integer type. If the pointer is null or valid, the integer result converted back to the pointer type shall compare equal to the original pointer.[98] For two valid pointer values that compare equal, conversion to the same integer type yields identical values.

8 A pointer to an object type may be converted to a pointer to a different object type with the same provenance. If the resulting pointer is not correctly aligned[99] for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type or **`void`** , the result ~~points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes~~ is the byte address of the object.

94) The obsolescent macro **`NULL`** is predefined as a null pointer constant, see 6.10.8.1, but new code should prefer the keyword **`nullptr`** wherever a null pointer constant is specified.

95) If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

96) Such a conversion happens implicitly when a pointer value is a controlling expression or when it is the operand of logical operators.

97) It is recommended that 0 is a member of that set.

98) Although such a round-trip conversion may be the identity for the pointer value, the side effect of exposing a storage instance still takes place.

99) In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

9 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

Forward references: the **`nullptr`** constant (6.4.4.5.2), cast operators (6.5.4), equality operators (6.5.10), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.17.1).

10 **NOTE** If the result `p` of an lvalue conversion or integer-to-pointer conversion is the end address of an exposed storage instance *A* and the start address of another exposed storage instance *B* that happens to follow immediately in the address space, a conforming program must only use one of these provenances in any expressions that is derived from `p`, see 6.2.5.

The following three cases determine if `p` is used with one of *A* or *B* and must hence not be used otherwise:

— Operations that constitute a use of `p` with either *A* or *B* and do not prohibit a use with the other:

- any relational operator or pointer subtraction where the other operand `q` may have both provenances, that is where `q` is also the result of a similar conversion and where `p == q`;
- `q == p` and `q != p` regardless of the provenance of `q`;
- addition or subtraction of the value 0;
- conversion to integer.

For the latter, *A* and *B* must have been exposed before, and so a any choice of provenance, that would otherwise have exposed one of the storage instances, is consistent with any other use.

— Operations that, if otherwise well defined, constitute a use of `p` with *A* and prohibit any use with *B*:

- Any relational operator or pointer subtraction where the other operand `q` has provenance *A* and cannot have provenance *B*.
- `p + n` and `p[n]`, where `n` is an integer strictly less than `0`.
- `p - n`, where `n` is an integer strictly greater than `0`.

— Operations that, if otherwise well defined, constitute a use of `p` with *B* and prohibit any use with *A*:

- Any relational operator or pointer subtraction where the other operand `q` has provenance *B* and cannot have provenance *A*.
- `p + n` and `p[n]`, where `n` is an integer strictly greater than `0`.
- `p - n`, where `n` is an integer strictly less than `0`.
- operations that access an object in *B*, that is indirection (`*p` or `p[n]` for `n == 0`) and member access (`p->member`).

6.3.2.4 `nullptr_t`

Constraint

1 A value of **`nullptr_t`** type shall not be converted to a type other than **`bool`** or a pointer type.

Description

2 When converted to **`bool`**, a value of type **`nullptr_t`** yields **`false`**. When converted to a pointer type, it yields a null pointer of that type.

6.4 Lexical elements

Syntax

1 *token:*

keyword
identifier
constant
string-literal
punctuator

preprocessing-token:

header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

Constraints

2 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.

Semantics

3 A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.[100] If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

4 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. There is one exception to this rule: header name preprocessing tokens are recognized only within **`#include`** preprocessing directives and in implementation-defined locations within **`#pragma`** directives. In such contexts, a sequence of characters that could be either a header name or a string literal is recognized as the former.

5 **EXAMPLE 1** The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating constant token), whether or not `E` is a macro name.

6 **EXAMPLE 2** The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression.

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5).

---

[100] An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files.

6.4.1 Keywords

Syntax

1 *keyword:* one of

`alignas`	`default`	~~`register`~~ `not`	`union`
`alignof`	`do`	~~`restrict`~~ `not_eq`	`unsigned`
`and`	`double`	`nullptr`	`void`
`and_eq`	`else`	`or`	`volatile`
`auto`	`enum`	`or_eq`	`while`
`bitand`	`extern`	`return`	~~`_Alignas`~~
`bitor`	`false`	`short`	~~`_Alignof`~~
`bool`	`float`	`signed`	~~`_Atomic`~~
`break`	`for`	`sizeof`	~~`_Bool`~~
`case`	`goto`	`static`	~~`_Complex`~~
`char`	`generic_selection`	`static_assert`	~~`_Generic`~~ `xor`
`compl`		`struct`	~~`_Imaginary`~~
`const`	`if`	`switch`	`xor_eq`
`constexpr`	`inline`	`thread_local`	`_Noreturn`
`continue`	`int`	`true`	~~`_Static_assert`~~
`decltype`	`long`	`typedef`	~~`_Thread_local`~~

Constraints

2 The keywords

`alignas`	`bitor`	`generic_selection`	`nullptr`	`true`
`alignof`	`bool`		`or_eq`	`xor_eq`
`and_eq`	`compl`		`or`	`xor`
`and`	`decltype`	`not_eq`	`static_assert`	
`bitand`	`false`	`not`	`thread_local`	

may optionally be predefined macro names (6.10.8.4). None of these shall be the subject of a `#define` or a `#undef` preprocessing directive.

Semantics

3 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords except in an attribute token, and shall not be used otherwise. ~~The~~

4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.[101)]

keyword ~~`_Imaginary` is reserved for specifying imaginary types~~	alternative spelling
`alignas`	`_Alignas`
`alignof`	`_Alignof`
`bool`	`_Bool`
`generic_selection`	`_Generic`
`static_assert`	`_Static_assert`
`thread_local`	`_Thread_local`

5 The spelling of keywords that are also predefined macros and that are subject to the `#` and `##` preprocessing operators is unspecified.[102)]

6 **NOTE** C also has optional imaginary types that are introduced with the keyword `_Imaginary`. This is rarely implemented in C and C++ has no equivalent for this, so this feature is not included in the C/C++ core. C also has the `restrict` and `_Atomic` qualifiers, which for the purpose of the C/C++ common core can be replaced by the `core::noalias` attribute and the `atomic_type` specifier, respectively. C's `register` keyword can also be replaced by using a `core::noalias` attribute, which can even be applied in a wider context, e.g. for file scope identifiers.

101) These alternative keywords are obsolescent features and should not be used for new code.

102) The intent of these specifications is to allow but not to force the implementation of the correspondig feature by means of a predefined macro.

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*

identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:

nondigit
universal-character-name
other implementation-defined characters

nondigit: one of

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

Semantics

2 An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

3 The use of universal character names in identifiers is specified in Annex D: Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in D.1.[103] The initial character shall not be a universal character name designating a character whose encoding falls into one of the ranges specified in D.2. An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation-defined.

4 When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword except in an attribute token.

Implementation limits

5 As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.

6 Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

Forward references: universal character names (6.4.3), macro replacement (6.10.3).

6.4.2.2 Predefined identifiers

[103]On systems in which linkers cannot accept extended characters, an encoding of the universal character name can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the `\u` in a universal character name. Extended characters can produce a long external identifier.

1 Several identifiers although they are not keywords are predefined and shall not be given a different definition by the program, be it by object, function, type or macro definitions. There are such identifiers of different categories, namely macros (6.10.8), constants (6.4.4.5), types (6.2.5.1) and objects (6.4.2.2.1).

6.4.2.2.1 Predefined objects

Semantics

1 The identifier `__func__` shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.[104]

2 This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5.

Recommended practice

3 Because C and C++ have a different instantiation model for inline functions, block scope static objects of such functions can be represented differently between the two languages. It is recommended that the control flow of applications is not made dependent of the comparison of the addresses of the `__func__` object(s) of an inline function, and that implementations issue a diagnostic whenever an attempt is made to expose the address of `__func__` beyond the function body of an inline function.

4 **EXAMPLE** Consider the code fragment:

```
#include <stdio.h>
void myfunc(void)
{
      printf("%s\n", __func__);
      /* ... */
}
```

Each time the function is called, it will print to the standard output stream:

```
myfunc
```

Forward references: the inline specifier (6.7.5), function definitions (6.9.1).

6.4.3 Universal character names

Syntax

1 *universal-character-name:*
  `\u` *hex-quad*
  `\U` *hex-quad hex-quad*

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

Constraints

2 ~~A~~ No universal character name shall be formed that specifies a short identifier that is not an ISO/IEC 10646 code point. Unless specified otherwise, no universal character name shall be formed that specifies a short identifier that is in the range D800 through DFFF inclusive.[105] Unless used

[104] Since the name `__func__` is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name `__func__`, the behavior is undefined.

[105] The notable exception is that four-digit short identifiers may be used with in UTF–16 wide strings to encode surrogate pairs.

within a character constant (6.4.4.4) or a string literal (6.4.5), a universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024~~($~~ ($), 0040~~(@~~ (@), or 0060~~('), nor one in the range D800 through DFFF inclusive~~ (`).[106]

Description

3 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Semantics

4 The universal character name `\U`*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*.[107] Similarly, the universal character name `\u`*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000*nnnn*).

5 **NOTE** C is a bit more restrictive than C++ about short identifiers that are less than 00A0. It extends the above interdictions for that range also to character constants and string literals. It seemed important to have one-to-one compatibility of such basic tools between C and C++ and to privilige usability. So this specification removes that restriction.

6.4.4 Constants

Syntax

1 *constant:*

integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

Constraints

2 Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

3 Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

1 *integer-constant:*

decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:

nonzero-digit
decimal-constant digit

octal-constant:

`0`
octal-constant octal-digit

hexadecimal-constant:

hexadecimal-prefix hexadecimal-digit
hexadecimal-constant hexadecimal-digit

[106] The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use as UTF–16 surrogates).

[107] Short identifiers for characters were first specified in ISO/IEC 10646–1:1993/Amd 9:1997.

hexadecimal-prefix: one of
`0x 0X`

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{\text{opt}}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{\text{opt}}$
long-long-suffix unsigned-suffix$_{\text{opt}}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

long-long-suffix: one of
`ll LL`

Description

2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

3 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix `0` optionally followed by a sequence of the digits `0` through `7` only. A hexadecimal constant consists of the prefix `0x` or `0X` followed by a sequence of the decimal digits and the letters `a` (or `A`) through `f` (or `F`) with values 10 through 15 respectively.

Semantics

4 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

5 The type of an integer constant is the first of the corresponding list in which its value can be represented.

Suffix	Decimal Constant	Octal or Hexadecimal Constant

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

6 If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

6.4.4.2 Floating constants

Syntax

1 *floating-constant:*
    decimal-floating-constant
    hexadecimal-floating-constant

decimal-floating-constant:
    fractional-constant exponent-part$_{\text{opt}}$ *floating-suffix*$_{\text{opt}}$
    digit-sequence exponent-part floating-suffix$_{\text{opt}}$

hexadecimal-floating-constant:
    hexadecimal-prefix hexadecimal-fractional-constant
        binary-exponent-part floating-suffix$_{\text{opt}}$
    hexadecimal-prefix hexadecimal-digit-sequence
        binary-exponent-part floating-suffix$_{\text{opt}}$

fractional-constant:
    digit-sequence$_{\text{opt}}$ **.** *digit-sequence*
    digit-sequence **.**

exponent-part:
    e *sign*$_{\text{opt}}$ *digit-sequence*
    E *sign*$_{\text{opt}}$ *digit-sequence*

sign: one of
    + -

digit-sequence:
  digit
  digit-sequence digit

hexadecimal-fractional-constant:
  hexadecimal-digit-sequence$_{\text{opt}}$ **.**
    hexadecimal-digit-sequence
  hexadecimal-digit-sequence **.**

binary-exponent-part:
  p *sign*$_{\text{opt}}$ *digit-sequence*
  P *sign*$_{\text{opt}}$ *digit-sequence*

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence hexadecimal-digit

floating-suffix:
  precision-suffix complex-suffix$_{\text{opt}}$
  complex-suffix precision-suffix$_{\text{opt}}$

precision-suffix: one of
  f l F L

complex-suffix: one of
  i I

Description

2 A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (**.**), followed by a digit sequence representing the fraction part. The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

Semantics

3 The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when **FLT_RADIX** is a power of 2, the result is correctly rounded.

4 An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**. If the suffix contains the letter **i** or **I**, the types are the corresponding complex types, and the value is a complex value with real part 0 and the value of the literal as imaginary part.

5 The values of floating constants may be represented in greater range and precision than that required by the type (determined by the suffix); the types are not changed thereby. See 5.2.4.2.2 regarding

evaluation formats.[108] The representation of the imaginary part of a complex floating constant shall be identical to the representation of the same real floating constant when `i` or `I` are omitted.

6 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. All floating constants of the same source form[109] shall convert to the same internal format with the same value.

Recommended practice

7 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.

8 The translation-time conversion of real floating constants should match the execution-time conversion of character strings by library functions, such as **`strtod`**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.[110]

6.4.4.3 Enumeration constants

Syntax

1 *enumeration-constant:*
    identifier

Semantics

2 An identifier declared as an enumeration constant has type **`int`**.

Forward references: enumeration specifiers (6.7.2.2).

6.4.4.4 Character constants

Syntax

1 *character-constant:*
    encoding-prefix$_{\text{opt}}$ **`'`** *c-char-sequence* **`'`**

encoding-prefix:
    `u8`
    `u`
    `U`
    `L`

c-char-sequence:
    c-char
    c-char-sequence c-char

c-char:
    any member of the source character set except
        the single-quote `'`, backslash `\`, or new-line character
    escape-sequence

escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence
    universal-character-name

[108] Hexadecimal floating constants can be used to obtain exact values in the semantic type that are independent of the evaluation format. Casts produce values in the semantic type, though depend on the rounding mode and may raise the inexact floating-point exception.

[109] `1.23`, `1.230`, `123e-2`, `123e-02`, and `1.23L` are all different source forms and thus need not convert to the same internal format and value.

[110] The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.22.1.3).

simple-escape-sequence: one of
`\' \" \? \\`
`\a \b \f \n \r \t \v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

Description

2 An *integer character constant* is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`. A UTF–*8 character constant* is the same, except prefixed by `u8`. A *wide character constant* is the same, except prefixed by the letter `L`, `u`, or `U`~~.~~; if it is prefixed by `u` it is a UTF–*16 character constant* and if by `U` it is a UTF–*32 character constant* . With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped ~~in an implementation-defined manner to members of the execution character set.~~ as described for translation phase 1 to a universal character name.

3 The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences:

single quote `'`	`\'`
double quote `"`	`\"`
question mark `?`	`\?`
backslash `\`	`\\`
octal character	`\`*octal digits*
hexadecimal character	`\x` *hexadecimal digits*

4 The double-quote `"` and question-mark `?` are representable either by themselves or by the escape sequences `\"` and `\?`, respectively, but the single-quote `'` and the backslash `\` shall be represented, respectively, by the escape sequences `\'` and `\\`.

5 The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

6 The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash `\` followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[111)]

[111)]The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See "future language directions" (6.11.4).

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

Prefix	Corresponding Type
none	`unsigned char`
`u8`	`unsigned char`
`L`	the unsigned type corresponding to `wchar_t`
`u`	`char16_t`
`U`	`char32_t`

10 A UTF–8, UTF–16, or UTF–32 character constant shall not contain more than one character.[112] ~~The value shall be~~ Such a character constant shall have any value that is representable with a single code unit of the corresponding encoding, in particular UTF~~–8 code unit .~~ –8 character constants shall be in the range 0 to 0x7F, UTF–16 character constants shall be in the range 0 to 0xFFFF, and UTF–32 character constants shall be in the ranges 0 to 0xD7FF or 0xE000 to 0x10FFFF.[113]

Semantics

11 An integer character constant has type **int**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., `'ab'`), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.

12 A UTF–8, UTF–16 or UTF–32 character constant has type ~~**unsigned char**.~~ **unsigned char**, **char16_t** or **char32_t**, respectively. The value of ~~a UTF–8~~ such a character constant is equal to its ISO/IEC 10646 code point value~~, provided that the code point value can be encoded as a single UTF–8 code unit~~.

13 A wide character constant prefixed by the letter **L** has type **wchar_t** ~~, an integer type defined in the header; a wide character constant prefixed by the letter **u** or **U** has type or , respectively, unsigned integer types defined in the header~~. The value of such a wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc** ~~, , or function as appropriate for its type,~~ function with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

14 **EXAMPLE 1** The construction `'\0'` is commonly used to represent the null character.

15 **EXAMPLE 2** Consider implementations that use eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant `'\xFF'` has the value −1; if type **char** has the same range of values as **unsigned char**, the character constant `'\xFF'` has the value +255.

16 **EXAMPLE 3** Even if eight bits are used for objects that have type **char**, the construction `'\x123'` specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are `'\x12'` and `'3'`, the construction `'\0223'` can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

17 **EXAMPLE 4** Even if 12 or more bits are used for objects that have type **wchar_t**, the construction `L'\1234'` specifies the implementation-defined value that results from the combination of the values `0123` and `'4'`.

Forward references: ~~common definitions (7.19),~~ the **mbtowc** function (7.22.8.2), Unicode utilities `<uchar.h>` (7.28).

[112] For example `u8'ab'` violates this constraint.

[113] This means that characters for UTF–8 encodings that need more than one byte can not be represented in a UTF–8 character constant. Also UTF–16 character constants may be surrogates, but UTF–32 character constants may not.

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:* one of

```
false    nullptr    true
```

Description

Some keywords represent constants of a specific value and type.

6.4.4.5.1 The `false` and `true` constants

Description

1 The keywords **`false`** and **`true`** represent constants of type **`bool`** that are suitable for use as are integer literals. Their values are 0 for **`false`** and 1 for **`true`**.[114] When used in preprocessor conditional expressions, the keywords **`false`** and **`true`** behave as if replaced with the pp-numbers `0` and `1`, respectively.[115]

2 **NOTE** Historically, C had the constants **`false`** and **`true`** with type **`int`**. This lead to unexpected results when used as arguments to type-generic interfaces and introduced an unfortunate incompatibility with C++. Users and implementations are invited to diagnose such situations, in particular where Boolean values (be they **`bool`** or **`int`**) are used in arithmetic other than array indexing.

6.4.4.5.2 The `nullptr` constant

Description

1 The keyword **`nullptr`** represents a null pointer constant of type **`nullptr_t`**. Unless specified otherwise, it is a suitable primary expression wherever a constant operand of pointer type is allowed for initialization, assignment, conversion, function argument, equality testing, the **`sizeof`** operator, logical operators, and as a controlling expression. If **`nullptr`** is used in any other context, the behavior is undefined.[116]

2 **NOTE** Because its type is underspecified, using **`nullptr`** as a controlling expression in a generic selection can lead to non-portable results.

Recommended practice

3 Implementations are encouraged to implement **`nullptr`** with a type that is not a scalar type, that is incompatible to any other type. They should diagnose the use of **`nullptr`**

— in any context where its use is undefined;

— as the controlling expression of a generic selection, unless that generic selection is itself not evaluated or the resulting type of the expression is independent of the effective choice;

— in a conversion to a type that is not a pointer type;

— as a second or third operand of a conditional operator if the other (second or third) operand has arithmetic type.

6.4.5 String literals

Syntax

1 *string-literal:*

encoding-prefix$_{\text{opt}}$ **"** *s-char-sequence*$_{\text{opt}}$ **"**

s-char-sequence:

s-char

[114] When used in arithmetic expressions after translation phase 4 the values of the keywords are promoted to type **`int`**.

[115] Therefore, arithmetic with **`false`** and **`true`** in translation phase 4 presents results that are generally consistent with later translation phases.

[116] In particular this prohibits the use of **`nullptr`** for any type of arithmetic operation, relational comparison, or in an operation that requires an lvalue.

s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote `"`, backslash `\`, or new-line character
escape-sequence

Constraints

2 A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`. A UTF–*8 string literal* is the same, except prefixed by **`u8`**. A *wide string literal* is the same, except prefixed by the letter **`L`**, **`u`**, or **`U`**; if it is prefixed by **`u`** it is a UTF–*16 string literal* and if by **`U`** it is a UTF–*32 string literal* .

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF–8 string literal) or a wide character constant (for a wide string literal), except that the single-quote `'` is representable either by itself or by the escape sequence `\'`, ~~but~~ the double-quote `"` shall be represented by the escape sequence `\"`~~.~~, and that a universal-character-name in a UTF–16 string literal may yield a surrogate pair.

Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[117] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **`char`**, and are initialized with the individual bytes of the multibyte character sequence. For UTF–8, UTF–16 and UTF–32 string literals, the array elements have type ~~**`char`**, and are initialized with the characters of the multibyte character sequence, as encoded in UTF–8. For wide string literals prefixed by the letter **`L`**, the array elements have type~~ **`char8_t`**, **`char16_t`** and **`char32_t`**, respectively, and are initialized with the ~~sequence of wide characters corresponding to the multibyte character sequence, as defined by the function with an implementation-defined current locale.~~ encoding sequence of the corresponding encoding. For wide string literals prefixed by the letter ~~**`u`** or **`U`**~~**`L`**, the array elements have type ~~or , respectively,~~ **`wchar_t`** and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by ~~successive calls to the , or function as appropriate for its type,~~ the **`mbstowcs`** function with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values.[118] If the program attempts to modify such an array, the behavior is undefined.

8 **EXAMPLE 1** This pair of adjacent character string literals

```
"\x12" "3"
```

[117] A string literal might not be a string (see 7.1.1), because a null character can be embedded in it by a `\0` escape sequence.
[118] This allows implementations to share storage instances for string literals and constant compound literals (6.5.2.5) with the same or overlapping representations.

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 **EXAMPLE 2** Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: ~~common definitions (7.19),~~ the **mbstowcs** function (7.22.9.1), Unicode utilities `<uchar.h>` (7.28).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

[ ] [[ ]] ( ) { } . ~~->~~ →
++ -- & * + - ~ ~~!~~ ¬
~~/ % << >> < > <= >= == != ^ | && ||~~ × / % ⊲⊠ ⊠⊳
<=> < > ≤ ≥ ≡ ≠ ∩ ^ ∪ ∧ ∨
? : ~~:: ; ...~~ :: ; ...
= ~~*=~~ ×= /= %= += -= ~~<<= >>= &= ^= |=~~ ⊲⊠= ⊠⊳= ∩= ^= ∪=
, # ##
<: :> <% %> -> ! << >> <= >= == != |
&& || :: ... *= <<= >>= &= |= %: %:%:

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

3 In all aspects of the language, ~~the six tokens~~[119)] in the the following table, for each line the token, *digraph* , and keyword, if any, behave the same except for their spelling. ~~<: :> <% %> %:~~[119)] For those tokens that map to a single Unicode character, the column labeled "code" shows the code point in ISO 10646 that corresponds to the token. When placed in a character string either directly or by applying the **#** operator (see 6.10.3.2) the resulting multi-byte sequence for the token shall be exactly the same as when the corresponding code is used as short identifier in a universal character name (see 6.4.3).

[119)] ~~These tokens are sometimes called "digraphs".~~
[119)] Thus [and <: behave differently when "stringized" , but can otherwise be freely interchanged.

token	digraph	keyword	code	token	digraph	keyword	code
[	<:		0x005b	≡	==		0x2261
]	:>		0x005d	≠	!=	not_eq	0x2260
(			0x0028	∩	&	bitand	0x2229
)			0x0029	^		xor	0x005e
{	<%		0x007b	∪	|	bitor	0x222a
}	%>		0x007d	∧	&&	and	0x2227
.			0x002e	∨	||	or	0x2228
→	->		0x2192	?			0x003f
++				:			0x003a
--				∷	::		0x2237
&			0x0026	;			0x003b
*			0x002a	…	...		0x2026
+			0x002b	=			0x003d
-			0x002d	×=	*=		
~		compl	0x007e	/=			
¬	!	not	0x00ac	%=			
×	*		0x00d7	+=			
/			0x002f	-=			
%			0x0025	⌫=	<<=		
⌫	<<		0x232b	⌦=	>>=		
⌦	>>		0x2326	∩=	&=	and_eq	
<=>				^=		xor_eq	
<			0x003c	∪=	|=	or_eq	
>			0x003e	,			0x002c
≤	<=		0x2264	#	%:		0x0023
≥	>=		0x2265	##	%:%:		

~~behave, respectively, the same as the six tokens [ ] { } # ##~~

~~except for their spelling.~~ Similarly, the token pairs [[and]] stand in for the tokens ⟦ (code 0x27e6) and ⟧ (code 0x27e7), respectively.

4 **NOTE 1** Currently, neither C nor C++ support all the four digit Unicode characters as punctuators. Nevertheless, using the digraphs can lead to lexical ambiguities because the same digraph may represent different tokens or because adjancent tokens may be merged, or not. C and C++ apply different strategies to resolve such ambiguities and so generally the use of digraphs should be avoided when writing programs for the C/C++ core. Therefore implementations that wish to serve the C/C++ core should offer support for these punctuators as extensions.

5 **NOTE 2** In C, the keywords that are listed in this table are only available as macros via the library header `<iso646.h>`. In contrast, in C++ they have been keywords since the beginning. Code that targets the C/C++ core must be able to deal with legacy code that uses these keywords so they were added to this specification.

Recommended practice

6 If they have to use digraphs, applications should avoid lexical ambiguities by adding white space around these digraphs.

Forward references: expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8).

6.4.7 Header names

Syntax

1 *header-name:*
 < *h-char-sequence* **>**
 " *q-char-sequence* **"**

h-char-sequence:
 h-char
 h-char-sequence h-char

h-char:
any member of the source character set except
the new-line character and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except
the new-line character and **"**

Semantics

2 The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2.

3 If the characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the < and > delimiters, the behavior is undefined. Similarly, if the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, the behavior is undefined.[120] Header name preprocessing tokens are recognized only within **`#include`** preprocessing directives and in implementation-defined locations within **`#pragma`** directives.[121]

4 **EXAMPLE** The following sequence of characters:

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a { on the left and a } on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

Forward references: source file inclusion (6.10.2).

6.4.8 Preprocessing numbers

Syntax

1 *pp-number:*
digit
. *digit*
pp-number digit
pp-number identifier-nondigit
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

Description

2 A preprocessing number begins with a digit optionally preceded by a period (`.`) and may be followed by valid identifier characters and the character sequences **`e+`**, **`e-`**, **`E+`**, **`E-`**, **`p+`**, **`p-`**, **`P+`**, or **`P-`**.

[120] Thus, sequences of characters that resemble escape sequences cause undefined behavior.
[121] For an example of a header name preprocessing token used in a **`#pragma`** directive, see 6.10.9.

3 Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

4 A preprocessing number does not have type or a value; it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[122)]

2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

3 **EXAMPLE**

```
"a//b"                    // four-character string literal
#include "//e"            // undefined behavior
// */                     // comment, not syntax error
f = g/**//h;              // equivalent to f = g / h;
//\
i();                      // part of a two-line comment
/\
/ j();                    // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();            // syntax error, not comment
/*//*/ l();               // equivalent to l();
m = n//**/o
  + p;                    // equivalent to m = n + p;
```

122)Thus, `/* ... */` comments do not nest.

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value,[123] or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[124]

3 The grouping of operators and operands is indicated by the syntax.[125] Except as specified later, side effects and value computations of subexpressions are unsequenced.[126]

4 Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

6 The *effective type* of an object for an access to its stored value and state is the declared type of the object, ~~if any~~unless that type is compatible to **void**[].[127] Otherwise, the object has a declared type that is compatible to **void**[]. If a value is stored into ~~an object having no declared type~~ such an object through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into ~~an object having no declared type~~ such an object using **memcpy** or **memmove**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one.[128] For all other accesses to ~~an object having no declared type~~such an object, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the

[123]Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

[124]This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

[125]The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.5.1), subscripting brackets `[]` (6.5.2.1), function-call parentheses `()` (6.5.2.2), and the conditional operator `?:` (6.5.16).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

[126]In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

[127]For the purpose of the determination of the effective type of an object with allocated storage duration behaves as if declared with a type compatible to **void**[].

[128]These provisions concerning the effective type not withstanding, the internal state of an opaque object or sub-object cannot be copied. Therefore a byte copy operation may bless an object with an effective type whereas the state of that object is still indeterminate.

following types:[129)]

- — a type compatible with the effective type of the object,
- — a qualified version of a type compatible with the effective type of the object,
- — a type that is the signed or unsigned type corresponding to the effective type of the object,
- — a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- — an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- — a character type.

8 A floating expression may be *contracted*, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[130)] The `FP_CONTRACT` pragma in `<math.h>` provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[131)]

9 **NOTE** C and C++ have very different strategies concerning the value category of expressions. Generally, for C++ expressions are *lvalues* whenever that is possible. In contrast to that, in C most operators undergo *lvalue conversion* (see 6.3.2.1) *before* they enter an expression and the information about the object(s) that entered into an expression is discarded. By that, a lot of expressions that are valid for C++ are not valid for C. *E.g* in C++ the prefix increment operator `++` can be applied multiple times in the same expression (`++ ++a`) or the ternary operator can be used on the left side of an assignment (`isit ? a : b)= 76;`. Both are invalid for C.

For C, lvalues only enter into expressions that are supposed to modify an object (such as assignment operators, increment and decrement), that compute its address (address-of operator), that access members (`.member` operator), or that query type properties such as size or alignment. The result of an expression is only an lvalue for the dereference operator `*` and for member access (`.` and `->`).

Programming for the C/C++ core implies not to use such constructs and we volutarily keep the possibility of returning lvalues out of this core specification.

Forward references: the `FP_CONTRACT` pragma (7.12.2), copying functions (7.24.2).

6.5.1 Primary expressions

Syntax

1 *primary-expression:*
  identifier
  constant
  string-literal
  (*expression* **)**
  generic-selection

Semantics

2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[132)]

3 A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4 A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

[129)] The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

[130)] The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions.

[131)] This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

[132)] Thus, an undeclared identifier is a violation of the syntax.

5 A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

6 A generic selection is a primary expression. Its type and value depend on the selected generic association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax

1 *generic-selection:*
~~**_Generic** **(** *assignment-expression*~~ **generic_selection** **(** *controlling-expression* **,** *generic-assoc-list* **)**

generic-assoc-list:
generic-association
generic-assoc-list **,** *generic-association*

generic-association:
type-name **:** *assignment-expression*
default **:** *assignment-expression*

controlling-expression:
expression

Constraints

2 The controlling expression shall be an assignment expression.[133]

3 A generic selection shall have no more than one **default** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. The type of the controlling expression is ~~the typeof the expression as if it had undergone an lvalue conversion,~~its generic type.[134] ~~array to pointer conversion, or function to pointer conversion.~~ That type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no **default** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

4 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **default** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

5 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, or a void expression if its result expression is, respectively, an lvalue, a function designator, or a void expression. A generic selection that is the operand of a **decltype** specification behaves as if the selected assignment expression had been the operand.[135]

~~The~~

6 **EXAMPLE 1** Provided there are functions **cbrt**, **cbrtf** and **cbrtl**, the **cbrt** type-generic macro could be implemented as follows:

```
        #define cbrt(X) _Generic((X),                    \
#define cbrt                                      \
```

[133] That means that it may not have a top level comma operator.

[134] ~~An lvalue conversion drops~~ A generic type has no qualifiers.

[135] Thus if the selected assignment expression is an identification *id*, the effect is as if the specifier had been given as **decltype**(*id*).

```
    [](auto x) {                    \
        return generic_selection(x, \
            long double: cbrtl,     \
            default: cbrt,          \
                        float: cbrtf            \
                        )(X)
            float: cbrtf)(x);       \
    }
```

Here the generic selection ensures that the correct function is chosen according to the inferred type of the paramenter `x` of the lambda. The englobing generic function literal has a return type that is the return type of the selected function. The function literal ensures that this version of the **`cbrt`** macro is converted to a function pointer when used outside a function call.

7 **EXAMPLE 2** A combination of a generic selection with a lambda may also be used to avoid to write several functions to implement a type generic functionality.

```
#define absconvert(X)                          \
    generic_selection((X),                     \
        char:             (unsigned char)+(X), \
        signed char:      (unsigned char)+(X), \
        signed short:    (unsigned short)+(X), \
        int:                   (unsigned)+(X), \
        long:             (unsigned long)+(X), \
        long long:   (unsigned long long)+(X), \
        default:                        +(X))

#define abs                           \
    [] (auto x) {                     \
        auto y = absconvert(x);       \
        if (x < 0) {                  \
            y = -absconvert(+x);      \
        }                             \
        return y;                     \
    }
```

~~See 7.25 how such a macro could be implemented with the required rounding properties.~~

6.5.2 Postfix operators

Syntax

1 *postfix-expression:*

primary-expression
~~*postfix-expression* **[** *expression* **]**~~
~~*postfix-expression* **(** *argument-expression-list*$_{opt}$ **)**~~
~~*postfix-expression* **.** *identifier*~~ *array-subscript*
~~*postfix-expression* **->** *identifier*~~ *function-call*
~~*postfix-expression* **++**~~ *member-access*
~~*postfix-expression* **--**~~ *postfix-addition*
~~**(** *type-name* **)** **{** *initializer-list* **}**~~ *compound-literal*
~~**(** *type-name* **)** **{** *initializer-list* **,** **}**~~ *lambda-expression*

6.5.2.1 Array subscripting

Syntax

1 ~~*argument-expression-list:*~~

~~*assignment-expression*~~ *array-subscript:*
~~*argument-expression-list* **,** *assignment-expression*~~ *postfix-expression* **[** *expression* **]**

Constraints

2 One of the expressions shall have type "pointer to complete object *type*", the other expression shall have integer type, and the result has type "*type*".

Semantics

3 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2` -th element of `E1` (counting from zero).

4 Successive subscript operators designate an element of a multidimensional array object. If `E` is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary `*` operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

5 **EXAMPLE** Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x`

is a 3×5 array of

~~`int`~~ `int`s; more precisely, `x` is an array of three element objects, each of which is an array of five ~~`int`~~ `int`s. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five ~~`int`~~ `int`s. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five `int` objects. The results are added and indirection is applied to yield an array of five ~~`int`~~ `int`s. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the ~~`int`~~ `int`s, so `x[i][j]` yields an ~~`int`~~ `int`.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.8.2).

6.5.2.2 Function calls

Syntax

1 *function-call:*
  postfix-expression **(** *argument-expression-list*$_{opt}$ **)**

argument-expression-list:
  assignment-expression
  argument-expression-list **,** *assignment-expression*

Constraints

2 The ~~expression that denotes the called function~~postfix expression[136] shall have ~~type~~ lambda type or pointer to function type, returning **`void`** or returning a complete object type other than an array or opaque type.

3 ~~If the expression that denotes the called function has a type that includes a prototype, the~~ The number of arguments shall agree with the number of parameters of the function or lambda type. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

[136] Most often, this is the result of converting an identifier that is a function designator.

Semantics

4 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function or lambda. The list of expressions specifies the arguments to the function or lambda.

5 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[137)]

6 If the expression that denotes the called function has lambda type or type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **`void`**.

7 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**. No such an argument shall be **`nullptr`**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (~~, ...~~, ...) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined.

8 If the expression that denotes the called function is a lambda or is a function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

9 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

10 If ~~the function is~~ a function is called that is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

11 If the expression that denotes the called function is a generic lambda, its prototype is completed by inferring types of underspecified parameters from the generic types of the corresponding arguments. This type inference for parameters takes place before any argument conversion or promotion. If necessary, the return type of the lambda is inferred accordingly, once the parameter types have been determined.

12 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function or lambda is indeterminately sequenced with respect to the execution of the called function.[138)]

13 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions or lambdas.

14 **EXAMPLE** In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Forward references: function declarators (6.7.8.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.17.1).

137) A function or lambda can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function or lambda can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

138) In other words, function executions do not "interleave" with each other.

6.5.2.3 Structure and union members

Syntax

1 *member-access:*

postfix-expression **.** *identifier*

postfix-expression → *identifier*

Constraints

2 The first operand of the . operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

3 The first operand of the ~~->~~ → operator shall have type "pointer to atomic, qualified, or unqualified structure" or "pointer to atomic, qualified, or unqualified union", and the second operand shall name a member of the type pointed to.

Semantics

4 A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member,[139] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

5 A postfix expression followed by the ~~->~~ → operator and an identifier designates a member of a structure or union object. The pointer value shall be valid, not be the end address of its provenance and be correctly aligned for the structure or union type. The value is that of the named member of the object to which the first expression points, and is an lvalue.[140] If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

6 Accessing a member of an atomic structure or union object results in undefined behavior.[141]

7 One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a *common initial sequence* if corresponding members have compatible types ~~(and, for bit-fields, the same widths)~~ for a sequence of one or more initial members.

8 **NOTE** C++ has a third notation to access a member of a structure (class) or union, but without refering to an object. This works with identifiers that are chained with a :: token. Translated into C an access as in the following

```
typedef struct A A;
typedef struct B B;
struct A { double a; };
struct B { A ba; };
...
sizeof(B::ba::a)
...
```

would be equivalent to

```
sizeof(((B){ }).ba.a)
```

[139] If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called "type punning"). This might be a trap representation.

[140] If &E is a valid pointer expression (where & is the "address-of" operator, which generates a pointer to its operand), the expression (&E) → MOS is the same as E.MOS.

[141] For example, a data race would occur if access to the entire structure or union in one thread conflicts with access to a member from another thread, where at least one access is a modification. Members can be safely accessed using a non-atomic object which is assigned to or from the atomic object.

that is, to create a compound literal of the requested type (the first element in the identifier chain, B) and then iteratively accessing the members of that compound literal (ba and a) with the . operator. C has no structure or union members that would be allowed in evaluations without having a concrete instance of such a type, and the use of such a construct would be restricted to contexts that are not evaluated, that is **sizeof**, **alignof**, and the controlling expression in a generic selection (plus **decltype** with this specification). Therefore, this feature seemed to be of minor importance for the common C/C++ core and was not added.

9 The usage of that feature is not conforming to the syntax of C and is therefore a constraint violation. All implementations that target the common C/C++ core should diagnose this use of the :: token.

10 **EXAMPLE 1** If f is a function returning a structure or union, and x is a member of that structure or union, f().x is a valid postfix expression but is not an lvalue.

11 **EXAMPLE 2** In:

```
struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;
```

the various members have the types:

```
s.i     int
s.ci    const int
cs.i    const int
cs.ci   const int
vs.i    volatile int
vs.ci   volatile const int
```

12 **EXAMPLE 3** The following is a valid fragment:

```
union {
        struct {
                int     alltypes;
        } n;
        struct {
                int     type;
                int     intnode;
        } ni;
        struct {
                int     type;
                double doublenode;
        } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
        if (sin(u.nf.doublenode) == 0.0)
if (u.n.alltypes ≡ 1)
        if (sin(u.nf.doublenode) ≡ 0.0)
                /* ... */
```

The following is not a valid fragment (because the union type is not visible within function f):

```
struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
        if (p1->m  <  0)
                p2->m = -p2->m;
        return p1->m;
        if (p1→m  <  0)
                p2→m = -p2→m;
        return p1→m;
}
int g()
```

```
    {
        union {
            struct t1 s1;
            struct t2 s2;
        } u;
        /* ... */
        return f(&u.s1, &u.s2);
    }
```

Forward references: address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1).

6.5.2.4 Postfix increment and decrement operators

Syntax

1 *postfix-addition:*
 postfix-expression **++**
 postfix-expression **--**

Constraints

2 The operand of the postfix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

3 The result of the postfix ++ operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation.~~Postfix ++ on an object with atomic type is a read-modify-write operation with **memory_order_seq_cst** memory order semantics.~~[142)]

4 The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

Forward references: additive operators (6.5.6), compound assignment (6.5.17.2).

6.5.2.5 Compound literals

Syntax

1 *compound-literal:*
 (*type-name* **)** **{** *initializer-list* **}**
 (*type-name* **)** **{** *initializer-list* **,** **}**

[142)]Where a pointer to an atomic object can be formed and E has integer type or pointer type, E++ is equivalent to the following code sequence where *A* is the type of E and C is the corresponding non-atomic, unqualified type:

```
    A *addr = &E;
    C old = *addr;
    C new;
    do {
        new = old + 1;
    } while (!atomic_compare_exchange_weak(addr, &old, new));
```

with old being the result of the operation.
Special care is necessary if E has floating type; see 6.5.17.2.

Constraints

2 The type name shall specify a complete object type or an array of unknown size, but not a variable length array type.

3 All the constraints for initializer lists in 6.7.12 also apply to compound literals.

Semantics

4 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.[143)]

5 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.12, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

6 The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

7 All the semantic rules for initializer lists in 6.7.12 also apply to compound literals.[144)]

8 String literals, and compound literals with const-qualified types, need not designate distinct objects.[145)]

9 **NOTE** C and C++ have quite different concepts of the lifetime of the object that are created by compound literals. Applications should constrain their usage to the full expression that contains them, see 6.2.4

10 **EXAMPLE 1** The file scope definition

```
int *p = (int []){2, 4};
```

initializes p to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

11 **EXAMPLE 2** In contrast, in

```
void f(void)
{
      int *p;
      /*...*/
      p = (int [2]){*p};
      /*...*/
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

12 **EXAMPLE 3** Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
      (struct point){.x=3, .y=4});
```

Or, if drawline instead expected pointers to **struct** point:

```
drawline(&(struct point){.x=1, .y=1},
      &(struct point){.x=3, .y=4});
```

13 **EXAMPLE 4** A read-only compound literal can be specified through constructions like:

143)Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or **void** only, and the result of a cast expression is not an lvalue.

144)For example, subobjects without explicit initializers are initialized to zero.

145)This allows implementations to share storage instances for string literals and constant compound literals with the same or overlapping representations.

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

14 **EXAMPLE 5** The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

15 **EXAMPLE 6** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
(const char []){"abc"} ≡ "abc"
```

might yield 1 if the literals' storage instance is shared.

16 **EXAMPLE 7** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

17 **EXAMPLE 8** Each compound literal creates only a single object in a given scope:

```
struct s { int i; };

int f (void)
{
      struct s *p = 0, *q;
      int j = 0;

again:
      q = p, p = &((struct s){ j++ });
      if (j  <  2) goto again;

      return p == q && q->i == 1;
      return p ≡ q ∧ q→i ≡ 1;
}
```

The function `f()` always returns the value 1.

18 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

Forward references: type names (6.7.9), initialization (6.7.12).

6.5.2.6 Lambda expressions

Syntax

1 *lambda-expression:*
capture-clause parameter-clause$_{opt}$ *attribute-specifier-sequence*$_{opt}$ *function-body*

capture-clause:
[*capture-list*$_{opt}$ **]**

capture-list:
=
identifier-capture

capture-list **,** *identifier-capture*

identifier-capture:
identifier
identifier **=** *assignment-expression*

parameter-clause:
(*parameter-type-list*$_{\text{opt}}$ **)**

Constraints

2 A lambda expression shall not be operand of the unary & operator.[146)]

3 The identifiers of identifier captures of the first form, if any, shall be names of complete objects with automatic storage duration that do not have opaque or array type and that are visible at the point of evaluation of the lambda expression. No identifier in an identifier capture shall appear in more than one such capture.

4 In addition to the identifiers in an identifier capture, the parameters that are declared in the parameter clause, if any, and identifiers that are declared in the function body, the function body shall only use identifiers according to the usual scoping rules, with the restriction that identifiers corresponding to objects with automatic storage duration shall only be evaluated within the assignment expression of an identifier capture, or if the capture clause starts with the token =.[147)]

5 A lambda expression for which at least one parameter declaration is underspecified has an opaque type. It shall only occur in a void expression, as the postfix expression of a function call or, if the capture clause is empty, in a conversion to a pointer to function with fully specified parameter types. For a void expression, it has no side effects and shall be ignored.

6 For a function call, the type of the parameters of a lambda shall be determined as follows. First, for each parameter, in order of declaration, a declared or inferred type is determined; if the parameter is underspecified that type is inferred from the call arguments analogous to 6.7.13, only that the inferred type for an array argument is the array type and that of a function specifier is the function type. Then, after the declared or inferred types of all parameters have been determined, analogously to function declarators (6.7.8.3) any array or function parameters are adjusted to possibly qualified pointer types. For a conversion of any arguments, the parameter types shall be those of the function type.

7 After determinining the generic type of the identifiers in the capture clause and then the parameter types, if any, the function body shall be such that a return type *type* of the function according to the rules in 6.9.1 can be inferred. If the lambda occurs in a conversion to a function pointer, the inferred return type shall be the same as the specified return type of the function pointer, if any.

Semantics

8 If the parameter clause is omitted, a clause of the form `()` is assumed. A lambda expression where the capture list in the capture clause is omitted is called a *function literal expression* or *function literal* . A lambda expression for which at least one parameter is underspecified is a *generic lambda* .

9 For each lambda expression, the return type *type* is inferred as indicated in the constraints. A lambda expression λ that is not generic defines an unspecified lambda type `L` that is the same for every evaluation of λ. If λ appears in a context that is not a function call, a value of type `L` is formed that identifies λ and the specific set of values of the identifiers in the capture clause for the evaluation, if any. This is called a *lambda value* . It is unspecified, whether two lambda expressions λ and κ share the same lambda type even if they are lexically equal but appear at different points of the program.

[146)] Objects with lambda type that can be operand of the unary & operator can be formed by type inference and initialization with a lambda value.

[147)] Identifiers of visible automatic objects that are not captured, may still be used if they are not evaluated, for example in **`decltype`** and **`sizeof`** (if they are not VM types) or as controlling expression of a generic primary expression.

10 An identifier `id` of an identifier capture is called a *capture* ; if the identifier capture is of the first form, a second form as if given as

```
id = id
```

is assumed. In particular, `id` must be the name of an object of automatic storage duration from a surrounding scope of the lambda expression. If the capture list starts with the token `=` and `id` is an identifier of a surrounding scope that corresponds to an object of automatic storage duration and that is used within the scope of the lambda expression (including the parameter list), the effect is as if `id` appeared as a capture.

11 The implicit or explicit assignment expression `E` in the identifier capture determines the type `T` which is the generic type of `E` and the value $\mathtt{E}_0$ of the capture within the whole scope of the lambda expressions (including the parameter list). The value $\mathtt{E}_0$ is determined immediately before the evaluation of the parameter list and the function body of the lambda expression. For each capture the effect is similar to as if as an additional parameter `id` of type **`const`** `T` is defined in front of the parameter list and if for every function call to the lambda value or any of its copies the same computed value $\mathtt{E}_0$ is provided as argument to `id`. If there is an object x in an outer scope to which the body of the lambda has no direct or indirect write access the address of a capture `id` and the address of x may be the same. If, within the function body, the address of the capture `id` or one of its members is taken, either explicitly by applying a unary `&` operator or by an array to pointer conversion,[148] and that address is used to modify the underlying object, the behavior is undefined.

12 For the value of a function literal that is not generic there is a function type `F` that corresponds to its return type and parameter types. The value of a function literal expression can be converted implicitly or explicitly (by a cast) to a pointer to `F` that identifies the value of the function literal. The resulting function pointer is the same for the whole program execution whenever a conversion of a lambda value corresponding to the same function literal expression is met.[149]

13 For a function literal expression that is generic and that is converted to a function type `F`, there shall be types for all underspecified parameters and a return type as in the constraint such that the adjusted function literal has type associated function type `F` as defined above.

14 If they are otherwise functionally equivalent, pointers to functions with internal linkage and pointers to functions converted from different function literals need not to be distinct.[150]

15 Other than for the scope of visibility of the corresponding identifier, the definition of an object with static storage duration within a lambda expression behaves as if the defined identifier is renamed uniquely for the whole translation unit, type inference is used to adjust the definition if necessary, and then the definition is moved before the (possibly nested) lambda expression.[151]

16 **NOTE** This specification opted to also apply and extend the C rules for parameter visibility from function declarationss to lambda expressions, such that parameter or capture names can be used as soon as they are declared. That possibility is important wherever there is a need to ensure consistency between types, array lengths or attributes.

17 **EXAMPLE 1** The following uses a function literal as a comparison function argument for **`qsort`**.

```
#define SORTFUNC(TYPE) [](size_t nmemb, TYPE A[nmemb]) {                  \
  qsort(A, nmemb, sizeof(A[0]),                                           \
        [](void const* x, void const* y){          /* comparison lambda  */ \
          TYPE X = *(TYPE const*)x;                                       \
          TYPE Y = *(TYPE const*)y;                                       \
          return (X < Y) ? -1 : ((X > Y) ? 1 : 0); /* return of type int */ \
```

148)The capture does not have array type, but if it has a union or structure type, one of its members may have such a type.

149)Thus a function literal has properties that are similar to a function declared with **`static`** and **`inline`**. A possible implementation of function literals is for them to have the type and value of the function pointer to which they convert.

150)In contrast to that, for a lambda expression that has captures each evaluation should be considered to result in a different lambda value, even if by coincidence the captured values are the same.

151)That means that with respect to definition and usage of local **`static`** objects, lambdas are more general than **`inline`** functions. This is because lambdas always have a well specified scope in which they are evaluated, that can also serve as scope of definition of such objects.

```
        }                                                                   \
        );                                                                  \
  return A;                                                                 \
  }
  ...
  long C[5] = { 4, 3, 2, 1, 0, };
  SORTFUNC(long)(5, C);                      // lambda → (pointer →) function call
  ...
  auto sortDouble = SORTFUNC(double);        // lambda value → lambda object
  double∗ (∗sF)(size_t nmemb, double[nmemb]) = sortDouble;   // conversion
  ...
  double∗ ap = sortDouble(4, (double[]){ 5, 8.9, 0.1, 99, });
  double B[27] = { /∗ some values ... ∗/ };
  sF(27, B);                                 // reuses the same function
  ...
  double∗ (∗sG)(size_t nmemb, double[nmemb]) = SORTFUNC(double); // conversion
```

This code evaluates the macro SORTFUNC twice, therefore in total four lambda expressions are formed.

The function literals of the "comparison lambdas" are not operands of a function call expression, and so by conversion a pointer to function is formed and passed to the corresponding call of **qsort**. Since the respective captures are empty, the effect is as if to define two comparison functions, that could equally well be implemented as **static** functions with auxiliary names and these names could be used to pass the function pointers to **qsort**.

The outer lambdas are again without capture. In the first case, for **long**, the lambda value is subject to a function call, and it is unspecified if the function call uses a specific lambda type or directly uses a function pointer. For the second, a copy of the lambda value is stored in the variable sortDouble and then converted to a function pointer sF. Other than for the difference in the function arguments, the effect of calling the lambda value (for the compound literal) or the function pointer (for array B) is the same.

For optimization purposes, an implementation may fold lambda values that are expanded at different points of the program such that effectively only one function is generated. For example here the function pointers sF and sG may or may not be equal.

18 **EXAMPLE 2** Even more, it is possible to implement a type-generic macro for sorting:

```
#define SORT [](size_t nmemb, auto A[nmemb]) {                              \
  qsort(A, nmemb, sizeof(A[0]),                                             \
        [](void const∗ x, void const∗ y){            /∗ comparison lambda  ∗/ \
          auto X = ∗(decltype(A))x;                                         \
          auto Y = ∗(decltype(A))y;                                         \
          return (X < Y) ? -1 : ((X > Y) ? 1 : 0); /∗ return of type int ∗/ \
        }                                                                   \
        );                                                                  \
  return A;                                                                 \
  }
  ...
  long C[5] = { 4, 3, 2, 1, 0, };
  SORT(5, C);
  ...
  double D[] = { 5, 8.9, 0.1, 99, };
  double∗ (∗sF)(size_t, double∗) = SORT;           // conversion
  ...
  double∗ ap = sF(4, D);
  double B[27] = { /∗ some values ... ∗/ };
  double∗ (∗sG)(size_t nmemb, double[nmemb]) = SORT; // conversion
  sG(27, B);
```

A can be used in a **decltype** specifier, because it is not evaluated, there. The SORT macro can then be used without providing further type specification to sort array C.

Assignment of the result of the SORT macro itself is not defined, because the two nested lambda values have insufficient type information. So to provide an instantiation for **double**, lambda expression is assigned to a function pointer. By that we obtain a fully specified parameter type list, and so the resulting function pointer can be kept in the sF variable.

Again, it is unspecified if the two function pointers sF and sG are identical or not.

19 **EXAMPLE 3** Consider the following generic function literal that computes the maximum value of two parameters X and Y.

```
#define MAXIMUM(X, Y)                                \
    [](auto a, auto b){                              \
      return (a < 0)                                 \
             ? ((b < 0) ? ((a < b) ? b : a) : b)     \
             : ((b ≥ 0) ? ((a < b) ? b : a) : a);    \
    }(X , Y)

    auto R = MAXIMUM(-1, -1U);
    auto S = MAXIMUM(-1U, -1L);
```

After preprocessing, the definition of R, becomes

```
auto R = [](auto a, auto b){
  return (a < 0)
     ? ((b < 0) ? ((a < b) ? b : a) : b)
     : ((b ≥ 0) ? ((a < b) ? b : a) : a);
  }(-1, -1U);
```

To determine type and value of R, first the type of the parameters in the function call are inferred to be **signed int** and **unsigned int**, respectively. With this information, the type of the **return** expression becomes the common arithmetic type of the two, which is **unsigned int**. Thus the return type of the lambda is that type. The resulting lambda value is the first operand to the function call operator (). So R has the type **unsigned int** and a value of **UINT_MAX**.

For S, a similar deduction shows that the value still is **UINT_MAX** but the type could be **unsigned int** (if **int** and **long** have the same width) or **long** (if **long** is wider than **int**).

As long as they are integers, regardless of the specific type of the arguments, the type of the expression is always such that the mathematical maximum of the values fits. So MAXIMUM implements a type generic maximum macro that is suitable for any combination of integer types.

20 **EXAMPLE 4**

```
void matmult(size_t k, size_t l, size_t m,
             double const A[k][l], double const B[l][m], double const C[k][m]) {
  // dot product with stride of m for B
  // ensure constant propagation of l and m
  auto λδ = [l,m](double const v[l], double const B[l][m], size_t m0) {
    double ret = 0.0;
    for (size_t i = 0; i < l; ++i) {
      ret += v[i]*B[i][m0];
    }
    return ret;
  };
  // vector matrix product
  // ensure constant propagation of l and m, and accessibility of λδ
  auto λμ = [l, m, λδ](double const v[l], double const B[l][m], double res[m]) {
    for (size_t m0 = 0; m0 < m; ++m0) {
      res[m0] = λδ(v, B, m0);
    }
  };
  for (size_t k0 = 0; k0 < k; ++k0) {
    double const (*Ap)[l] = A[k0];
    double (*Cp)[m] = C[k0];
    λμ(*Ap, B, *Cp);
  }
}
```

This function evaluates two lambda expressions with captures; $\lambda\delta$ has a return type of **double**, $\lambda\mu$ of **void**. Both lambda values serve repeatedly as first operand to function evaluation but the evaluation of the captures is only done once for each of the lambda expressions. For the purpose of optimization, an implementation could generate copies of the underlying functions for each evaluation of such a lambda expression such that the values of the captures `l` and `m` are replaced on a machine instruction level.

6.5.3 Unary operators

Syntax

1 *unary-expression:*

postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-name* `)`
~~`_Alignof`~~ `alignof (` *type-name* `)`

unary-operator: one of
`&  *  +  -  ~` ~~`!`~~ `¬`

6.5.3.1 Prefix increment and decrement operators

Constraints

1 The operand of the prefix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

2 The value of the operand of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix `--` operator is analogous to the prefix `++` operator, except that the value of the operand is decremented.

Recommended practice

4 C and C++ differ by the result category for these operators. Whereas for C they are values (and in this aspect equivalent to the corresponding postfix operator), in C++ they are lvalues and so they can be chained. Applications that target the C/C++ core should avoid the usage of these operators as operands to other expressions.

Forward references: additive operators (6.5.6), compound assignment (6.5.17.2).

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary `&` operator shall be either a function designator, the result of a `[]` or unary `*` operator, or an lvalue that designates an object~~that is not a bit-field and is not declared with the `register` storage-class specifier~~.[152)]

2 The operand of the unary `*` operator shall have pointer type.

Semantics

3 The unary `&` operator yields the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary `*` operator, neither that operator nor the `&` operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary `*` that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator. Otherwise, the result is a pointer to the object or function designated by its operand.

152) The `core::noalias` attribute may be used to inhibit the application of the unary `&` operator to objects and functions.

4 The unary * operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". ~~If an invalid value has been assigned to the pointer , the behavior of the unary * operator is undefined~~The pointer value shall be valid, not be the end address of its provenance and be correctly aligned for "*type*".[153]

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

6.5.3.3 Unary arithmetic operators

Constraints

1 The operand of the unary `+` or `-` operator shall have arithmetic type; of the `~` operator, integer type; of the ~~`!`~~ ¬ operator, scalar type.

Semantics

2 The result of the unary `+` operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

3 The result of the unary `-` operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

4 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

5 The ~~result of the~~ logical negation operator ~~`!`~~ ¬ ~~is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.~~ first converts the operand to **`bool`**. If that conversion yields **`true`**, the result is **`false`**; otherwise, the result is **`true`**.

6 **NOTE** In the current C specification the result of logical negation operator ¬ is not **`bool`** but **`int`**. Therefore it should not be used directly as argument to a type-generic macro or in another context that is sensible to the type of the expression.

6.5.3.4 The `sizeof` and `alignof` operators

Constraints

1 The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, or to the parenthesized name of such a type~~, or to an expression that designates a bit-field member. The **`_Alignof`**~~. The **`alignof`** operator shall not be applied to a function type or an incomplete type.

Semantics

2 The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

3 The ~~**`_Alignof`**~~ **`alignof`** operator yields the alignment requirement of its operand type. The operand is not evaluated and the result is an integer constant. When applied to an array type, the result is the alignment requirement of the element type.

4 When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[154] When applied to an operand that has structure or

[153] Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `*&E` is a function designator or an lvalue equal to `E`. If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, the address of an object after the end of its lifetime, or any other indeterminate value.

[154] When applied to a parameter declared to have array or function type, the **`sizeof`** operator yields the size of the adjusted

union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is **size_t** ~~, defined in (and other headers)~~.

6 **EXAMPLE 1** A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the alloc function presumably ensures that its return value is aligned suitably for conversion to a pointer to **double**.

7 **EXAMPLE 2** Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

8 **EXAMPLE 3** In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
      char b[n+3];       // variable length array
      return sizeof b;   // execution time sizeof
}

int main()
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

Forward references: ~~common definitions (7.19),~~ declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.9), array declarators (6.7.8.2).

6.5.4 Cast operators

Syntax

1 *cast-expression:*

unary-expression
(*type-name* **)** *cast-expression*

Constraints

2 Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type.

3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.17.1, shall be specified by means of an explicit cast.

4 A pointer type shall not be converted to any floating type. A floating type shall not be converted to any pointer type.

(pointer) type (see 6.9.1).

Semantics

5 Preceding an expression by a parenthesized type name converts the value of the expression to the unqualified version of the named type. This construction is called a *cast*.[155] A cast that specifies no conversion has no effect on the type or value of an expression.

6 If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

Forward references: equality operators (6.5.10), function declarators (6.7.8.3), simple assignment (6.5.17.1), type names (6.7.9).

6.5.5 Multiplicative operators

Syntax

1 *multiplicative-expression:*

cast-expression
multiplicative-expression ~~`*`~~ × *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

Constraints

2 Each of the operands shall have arithmetic type. The operands of the `%` operator shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary ~~`*`~~ × operator is the product of the operands.

5 The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

6 When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[156] If the quotient `a/b` is representable, the expression ~~`(a/b)*b + a%b`~~ `(a/b)×b + a \%b` shall equal `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

6.5.6 Additive operators

Syntax

1 *additive-expression:*

multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

Constraints

2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3 For subtraction, one of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible complete object

[155] A cast does not yield an lvalue.
[156] This is often called "truncation toward zero".

types; or

— the left operand is a pointer to a complete object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

Semantics

4 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

5 The result of the binary `+` operator is the sum of the operands.

6 The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. ~~In other words, if the expression `P` points to the i-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object.~~ If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated. The result pointer has the same provenance as the pointer operand.[157)]

9 When two pointers are subtracted, both shall be valid. If they compare equal the result is 0. Otherwise they shall have the same provenance and point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **`ptrdiff_t`** ~~defined in the header~~. If the result is not representable in an object of that type, the behavior is undefined. ~~In other words, if the~~

10 **NOTE 1** If the expression `P` points to the i-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object.

11 **NOTE 2** If the expressions `P` and `Q` point to, respectively, the i-th and j-th elements of an array object, the expression `(P)-(Q)` has the value $i-j$ provided the value fits in an object of type **`ptrdiff_t`**. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.

12 **NOTE 3** Another way to approach pointer arithmetic is first to convert the pointer(s) to character or **`void`** pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character or **`void`** pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which can overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

13 **EXAMPLE** Pointer arithmetic is well defined with pointers to variable length array types.

```
    {
```

[157)] If the pointer operand `P` had been the result of an integer-to-pointer or **`scanf`** conversion that could have two possible provenances, and the integer value added or subtracted is not 0, the provenance S for the additive operation (and henceforth other operations with `P`) must be such that the result lies in S (or one beyond).

```
        int n = 4, m = 3;
        int a[n][m];
        int (*p)[m] = a;  // p == &a[0]
        p += 1;           // p == &a[1]
        (*p)[2] = 99;     // a[1][2] == 99
        n = p - a;        // n == 1
        int (*p)[m] = a;  // p ≡ &a[0]
        p += 1;           // p ≡ &a[1]
        (*p)[2] = 99;     // a[1][2] ≡ 99
        n = p - a;        // n ≡ 1
    }
```

14 If array `a` in the above example were declared to be an array of known constant size, and pointer `p` were declared to be a pointer to an array of the same known constant size (pointing to `a`), the results would be the same.

Forward references: array declarators (6.7.8.2)~~, common definitions (7.19).~~ .

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*

additive-expression
shift-expression ~~<<~~ ⌫ *additive-expression*
shift-expression ~~>>~~ ⌦ *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of ~~`E1 << E2`~~ `E1 ⌫ E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is $\texttt{E1} \times 2^{\texttt{E2}}$, reduced modulo one more than the maximum value representable in the result type. If `E1` has a signed type and nonnegative value, and $\texttt{E1} \times 2^{\texttt{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of ~~`E1 >> E2`~~ `E1 ⌦ E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\texttt{E1}/2^{\texttt{E2}}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

6.5.8 Three-way comparison operator

Syntax

1 *compare-expression:*

shift-expression
compare-expression **<=>** *shift-expression*

Constraints

2 The three-way comparison operator shall only be used after the `<stdcompare.h>` header has been included. Additional constraints apply as specified in the correponding clause 7.31.

Description

3 The three-way comparison operator provides a tool to compare values of most object or function type. For the details see the indicated clause.

6.5.9 Relational operators

Syntax

1 *relational-expression:*

> ~~*shift-expression*~~ *compare-expression*
> *relational-expression* **<** ~~*shift-expression*~~ *compare-expression*
> *relational-expression* **>** ~~*shift-expression*~~ *compare-expression*
> *relational-expression* ~~**<=**~~ ~~*shift-expression*~~ ≤ *compare-expression*
> *relational-expression* ~~**>=**~~ ~~*shift-expression*~~ ≥ *compare-expression*

Constraints

2 One of the following shall hold:

— both operands have real type; or

— both operands are pointers to qualified or unqualified versions of compatible object types.

Semantics

3 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

4 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 When two pointers are compared, ~~the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`. In all other cases, the behavior is undefined~~they shall both be valid and have the same provenance. The result depends on the relative ordering of their abstract addresses.

6 Each of the operators < (less than), > (greater than), ~~<=~~ ≤ (less than or equal to), and ~~>=~~ ≥ (greater than or equal to) shall yield ~~1~~ **`true`** if the specified relation ~~is true and 0 if it is false~~holds and **`false`** otherwise.[158] ~~The result has type **`int`**.~~

7 **NOTE** In the current C specification the result is not **`bool`** but **`int`**. Therefore it should not be used directly as argument to a type-generic macro or in another context that is sensible to the type of the expression.

6.5.10 Equality operators

Syntax

1 *equality-expression:*

> *relational-expression*
> *equality-expression* ~~**==**~~ ≡ *relational-expression*
> *equality-expression* ~~**!=**~~ ≢ *relational-expression*

Constraints

2 One of the following shall hold:

— both operands have arithmetic type;

[158] The expression `a<b<c` is not interpreted as in ordinary mathematics. As the syntax indicates, it means `(a<b)<c`; in other words, "if `a` is less than `b`, compare 1 to `c`; otherwise, compare 0 to `c`".

— both operands are pointers to qualified or unqualified versions of compatible types;

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**; or

— one operand is a pointer and the other is a null pointer constant.

Semantics

3 The ~~==~~ ≡ (equal to) and ~~!=~~ ≠ (not equal to) operators are analogous to the relational operators except for their lower precedence.[159] None of the operands shall be indeterminate. Each of the operators yields ~~1~~ **true** if the specified relation ~~is true and 0 if it is false. The result has type **int**~~ holds and **false** otherwise. For any pair of operands, exactly one of the relations ~~is true~~yields **true**.

4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

5 If both of the operands are null pointer constants, they compare equal.

6 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.

7 ~~Two pointers~~ If one operand is null they compare equal if and only if ~~both are nullpointers, both~~ the other operand is null. Otherwise, if both operands are pointers to ~~the same object (including a pointer to an object and a subobject at its beginning) or function, both~~ function type they compare equal if and only if they refer to the same function. Otherwise, they are pointers to ~~one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the addressspace.~~ objects and compare equal if and only if they have the same abstract address.

8 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

9 NOTE In the current C specification the result is not **bool** but **int**. Therefore it should not be used directly as argument to a type-generic macro or in another context that is sensible to the type of the expression.

6.5.11 Bitwise AND operator

Syntax

1 *AND-expression:*

equality-expression

AND-expression ~~&~~ ∩ *equality-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary ~~&~~ ∩ operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

[159] Because of the precedences, `a<b` ≡ `c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

6.5.12 Bitwise exclusive OR operator

Syntax

1 *exclusive-OR-expression:*
  AND-expression
  exclusive-OR-expression **^** *AND-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

6.5.13 Bitwise inclusive OR operator

Syntax

1 *inclusive-OR-expression:*
  exclusive-OR-expression
  inclusive-OR-expression ~~|~~ ∪ *exclusive-OR-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the ~~|~~ ∪ operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

6.5.14 Logical AND operator

Syntax

1 *logical-AND-expression:*
  inclusive-OR-expression
  logical-AND-expression ~~**&&**~~ ∧ *inclusive-OR-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The ~~&&~~ ∧ operator shall yield ~~1~~ `true` if both of its operands ~~compare unequal to 0~~yield `true` when converted to `bool`; otherwise, it yields ~~0.~~ `false`. The result has type ~~`int`~~ `bool`.

4 Unlike the bitwise binary ~~&~~ ∩ operator, the ~~&&~~ ∧ operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand ~~compares equal to 0~~converts to `false`, the second operand is not evaluated.

5 **NOTE** In the current C specification the result is not `bool` but `int`. Therefore it should not be used directly as argument to a type-generic macro or in another context that is sensible to the type of the expression.

6.5.15 Logical OR operator

Syntax

1 *logical-OR-expression:*
  logical-AND-expression

logical-OR-expression ~~||~~ ∨ *logical-AND-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The ~~||~~ ∨ operator shall yield ~~1~~ `true` if either of its operands ~~compare unequal to 0~~yields `true` when converted to `bool`; otherwise, it yields ~~0.~~ `false`. The result has type ~~`int`~~ `bool`.

4 Unlike the bitwise ~~|~~ ∪ operator, the ~~||~~ ∨ operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand ~~compares unequal to 0~~converts to `true`, the second operand is not evaluated.

5 **NOTE** In the current C specification the result is not `bool` but `int`. Therefore it should not be used directly as argument to a type-generic macro or in another context that is sensible to the type of the expression.

6.5.16 Conditional operator

Syntax

1 *conditional-expression:*

logical-OR-expression

logical-OR-expression **?** *expression* **:** *conditional-expression*

Constraints

2 The first operand shall have scalar type.

3 One of the following shall hold for the second and third operands:

— both operands have arithmetic type;

— both operands have the same structure or union type;

— both operands have void type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— both operands are `nullptr`;

— one operand is a pointer and the other is a null pointer constant; or

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`.

Semantics

4 The first operand is evaluated and the result is converted to `bool`; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first ~~compares unequal to 0~~converts to `true`; the third operand is evaluated only if the first ~~compares equal to 0~~converts to `false`; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.[160)]

5 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

[160)] A conditional expression does not yield an lvalue.

6 If both the second and third operands are **nullptr** the result has the same type and value as **nullptr**. Otherwise, if either of the second or third operands is **nullptr**, and the other is an integer constant expression of value 0 the behavior is undefined.[161]

7 If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**.

Recommended practice

8 C and C++ differ by the result category for this operator. Whereas for C it is a value, in C++ it may be an lvalue and so a conditional operator may for example form the left argument of an assignment. Applications that target the C/C++ core should avoid a usage of the conditional operator in places where a modifiable lvalue is required.

9 **EXAMPLE** The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

10 Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

`c_vp`	`c_ip`	`const void *`
`v_ip`	`0`	`volatile int *`
`c_ip`	`v_ip`	`const volatile int *`
`vp`	`c_cp`	`const void *`
`ip`	`c_ip`	`const int *`
`vp`	`ip`	`void *`

6.5.17 Assignment operators

Syntax

1 *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`=` ~~`*=`~~ `×=` `/=` `%=` `+=` `-=` ~~`<<=` `>>=` `&=` `^=` `|=`~~ `◁⊠=` `⊠▷=` `∩=` `^=` `∪=`

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. If a non-null pointer is stored by an assignment operator, either directly or within a structure or union object, the stored pointer object has the same provenance as the original. An assignment expression has the value of the left operand after the assignment,[162] but is not an lvalue. The type of an assignment

[161] If the other operand has arithmetic type but is not constant and 0, a constraint is violated.

[162] The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

Recommended practice

4 C and C++ differ by the result category for these operators. Whereas for C it is a value, in C++ it may be an lvalue and so these operators may be chained from left to right such as in `(a+=6)*=35` which is a constraint violation in C. Applications that target the C/C++ core should avoid a usage of assignment operators in places where a modifiable lvalue is required.

5 Implementations that conform to this specification should diagnose usages of these operators that are erroneous in one of the two languages.

6.5.17.1 Simple assignment

Constraints

1 One of the following shall hold:[163]

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of **void**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand is an atomic, qualified, or unqualified pointer, and the right is a null pointer constant; or

— the left operand has type atomic, qualified, or unqualified ~~**_Bool**~~ **bool**, and the right is a pointer.

Semantics

2 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

4 **EXAMPLE 1** In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
if ((c = f()) ≡ -1)
        /* ... */
```

the **int** value returned by the function could be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and

163)The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **int volatile * const**).

char is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable c would be declared as **int**.

5 **EXAMPLE 2** In the fragment:

```
char c;
int i;
long l;

l = (c = i);
```

the value of i is converted to the type of the assignment expression c = i, that is, **char** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **long int** type.

6 **EXAMPLE 3** Consider the fragment:

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;       // constraint violation
*cpp = &c;      // valid
*p = 0;         // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object c.

6.5.17.2 Compound assignment

Constraints

1 For the operators += and -= only, either the left operand shall be an atomic, qualified, or unqualified pointer to a complete object type, and the right shall have integer type; or the left operand shall have atomic, qualified, or unqualified arithmetic type, and the right shall have arithmetic type.

2 For the other operators, the left operand shall have atomic, qualified, or unqualified arithmetic type, and (considering the type the left operand would have after lvalue conversion) each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

Semantics

3 A *compound assignment* of the form E1 *op*= E2 is equivalent to the simple assignment expression E1 = E1 *op* (E2), except that the lvalue E1 is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. ~~If E1 has an atomic type, compound assignment is a read-modify-write operation with **memory_order_seq_cst** memory order semantics.~~

4 **NOTE** Where a pointer to an atomic object can be formed and E1 and E2 have integer or pointer type, this is ~~equivalent~~ similar to the following code sequence where ~~*T1is the*~~ *A1* is the type of E1, *C1* is the corresponding non-atomic and unqualified type of E1 ~~and~~, and ~~*T2*~~*C2* is the non-atomic and unqualifid type of E2:

```
T1 *addr = &E1;
T2 val = (E2);
T1 old = *addr;
T1 new;
A1 *addr = &E1;
C2 val = (E2);
C1 old = *addr;
C1 new;
do {
      new = old op val;
} while (!atomic_compare_exchange_strong(addr, &old, new));
} while (!atomic_compare_exchange_weak(addr, &old, new));
```

with new being the result of the operation. The difference is that if the combination of the values of old and val is invalid for the operation, there will no signal raised or trap performed. In particular:

— If "`old` *op* `val`" has a signed type and produces an overflow, `new` is the corresponding modulo of the mathematical result of the operation.
— If the value of `val` is invalid for *op*, the value of `new` is unspecified.
— If `C2` is a pointer type and the value of `old` is null or is indeterminate, the value of `new` is unspecified.
— If `C2` is a pointer type and the value of "`old` *op* `val`" would be indeterminate, the value of `new` is unspecified.

If `E1` or `E2` has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of `new` would also be discarded in order to satisfy the equivalence of `E1` *op*`=` `E2` and `E1 = E1` *op* `(E2)`. For example, if Annex F is in effect, the floating types involved have IEC 60559 formats, and `FLT_EVAL_METHOD` is 0, the equivalent code would be:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* ... */
        fenv_t fenv;
        T1 *addr = &E1;
        T2 val = E2;
        T1 old = *addr;
        T1 new;
        A1 *addr = &E1;
        C2 val = E2;
        C1 old = *addr;
        C1 new;
        feholdexcept(&fenv);
        for (;;) {
                new = old op val;
                if (atomic_compare_exchange_strong(addr, &old, new))
                        break;
                feclearexcept(FE_ALL_EXCEPT);
        }
        feupdateenv(&fenv);
```

If `FLT_EVAL_METHOD` is not 0, then ~~*T2*~~*C2* is expected to be a type with the range and precision to which `E2` is evaluated in order to satisfy the equivalence.

6.5.18 Comma operator

Syntax

1 *expression:*
assignment-expression
expression **,** *assignment-expression*

Semantics

2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.

Recommended practice

3 C and C++ differ by the result category for this operator. Whereas for C it is a value, in C++ it may be an lvalue and so this operator may be chained from left to right such as in `(f(),a)=0` which is a constraint violation in C. Generally, the use of the comma operator is often problematic because it can easily be mixed up with other usages of the comma punctuator, such as in function arguments, type-generic expressions or initializations.

4 Applications that target the C/C++ core should avoid a usage of the comma operator in places where a modifiable lvalue is required. Implementations that conform to this specification should diagnose usages of the comma operator that are erroneous in one of the two languages.

5 **EXAMPLE** As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
    f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Forward references: initialization (6.7.12).

6.6 Constant expressions

Syntax

1 *constant-expression:*
  conditional-expression

Description

2 A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated~~.~~,[164] or if they are a **`constexpr`** function call that fulfills the corresponding constraints or are contained in such a call.

4 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

5 No **`constexpr`** object or function call (see below) shall be formed that has a pointer type, unless it has a null pointer value or the value is the result of a cast of an integer constant expression to the pointer type.

Semantics

6 An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic range and precision shall be at least as great as if the expression were being evaluated in the execution environment.[165]

7 A ***`constexpr`*** *object* is a such declared object, or one of the elements or members of such an object, even recursively, such that any element or member designator only uses integer constant expressions, if any. A ***`constexpr`*** *function call* is a call that uses a function designator or lambda value that has the **`constexpr`** specifier and that fulfills the constraints of such a call in the context of a constant expression.

8 An *integer constant expression*[166] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **`constexpr`** objects or function calls of integer type, **`sizeof`** expressions whose results are integer constants, ~~**`_Alignof`**~~ **`alignof`** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **`sizeof`** or ~~**`_Alignof`**~~ **`alignof`** operator.

9 More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:

— a **`constexpr`** object or function call,

— an arithmetic constant expression,

— a null pointer constant,

— an address constant, or

— an address constant for a complete object type plus or minus an integer constant expression.

164) The operand of a **`decltype`**, **`sizeof`** or **`alignof`** operator is usually not evaluated (6.5.3.4).

165) The use of evaluation formats as characterized by **`FLT_EVAL_METHOD`** also applies to evaluation in the translation environment.

166) An integer constant expression is required in a number of contexts such as the ~~size of a bit-field member of a structure, the~~ value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

10 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, **constexpr** objects or function calls of arithmetic type, **sizeof** expressions whose results are integer constants, and ~~**_Alignof**~~ **alignof** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **sizeof** or ~~**_Alignof**~~ **alignof** operator.

11 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript [] and member-access . and ~~->~~ → operators, the address & and indirection * unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

12 An implementation may accept other forms of constant expressions.

13 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.

14 **EXAMPLE 1** In the following

```
    static unsigned int i = UINT_MAX ∨ 1 / 0;  // valid
    static unsigned int j = UINT_MAX ∪ 1 / 0;  // invalid
```

the initializer expression for `i` is a valid integer constant expression with value one, since only the first operand of the ∨ operator is evaluated. For `j` it is invalid because both operands are needed for a successful evaluation of the bitwise ∪ operator, even though any valid value for the second operand would lead to the same result.

15 **EXAMPLE 2** **constexpr** objects may have aggregate or union type:

```
struct string32 { size_t len; char str[32]; };
constexpr struct string32 capital = {
      .len = sizeof("ABCDEFGHIJKLMNOPQRSTUVWXYZ")-1,
      .str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
};
constexpr         char encodingA = capital.str[0];              // internal linkage
inline const      char encodingB = capital.str[1];              // external linkage
inline constexpr char encodingC = capital.str[2];              // same
static const char*const emptyness = &capital.str[capital.len]; // valid
constexpr     char*       weirdness = &capital.str[capital.len]; // constraint violation
inline const char*const uglyness  = &capital.str[capital.len]; // constraint violation
```

Here, the initializers of `encodingA`, `encodingB` and `encodingC` only use member access operators with integer constant expressions, so they are valid. As a result, they hold the representation value for the capital letters A, B and C, respectively, in the execution character set. They are themselves **constexpr** objects and evaluate to integer constant expressions. Therefore they may be used in any context where such a constant is allowed.

16 The evaluation of `capital.len` leads to an integer constant expression, taking the address in the initializer of `emptyness` is then valid and evaluates to the address of the terminating null character of the string. Thus the definition is valid. In that definition, **static** could not be replaced by **inline** or **constexpr** because the unary & operator is not valid for the initialization of a **constexpr** object with pointer value. Therefore the initialization of `weirdness` and `uglyness` are invalid and must be diagnosed.

Forward references: array declarators (6.7.8.2), the **constexpr** specifier (6.7.4), initialization (6.7.12).

6.7 Declarations

Syntax

1 *declaration:*

declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**
attribute-specifier-sequence declaration-specifiers init-declarator-list **;**
static_assert-declaration
attribute-declaration

declaration-specifiers:

declaration-specifier attribute-specifier-sequence$_{\text{opt}}$
declaration-specifier declaration-specifiers

declaration-specifier:

storage-class-specifier
type-specifier-qualifier
~~*function-specifier*~~ **`constexpr`**
`inline`
`_Noreturn`

init-declarator-list:

init-declarator
init-declarator-list **,** *init-declarator*

init-declarator:

declarator
declarator **=** *initializer*

attribute-declaration:

attribute-specifier-sequence **;**

Constraints

2 A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

5 A declaration specifies the interpretation and properties of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:

— for an object, causes ~~storage~~ a unique storage instance to be reserved for that object;

— for a function, includes the function body;[167)]

— for an enumeration constant, is the (only) declaration of the identifier;

— for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence, that indicate the linkage, storage duration, and part of the type of the entities that

[167)]Function definitions have a different syntax, described in 6.9.1.

the declarators denote. The init declarator list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared. The optional attribute specifier sequence appertains to each of the entities declared by the declarators of the init declarator list.

7 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters, it is the adjusted type (see 6.7.8.3) that is required to be complete.

8 The optional attribute specifier sequence terminating a sequence of declaration specifiers appertains to the type determined by the preceding sequence of declaration specifiers. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

9 Except where specified otherwise, the meaning of an attribute declaration is implementation-defined.

10 In some situations a stricter correspondence of types, qualifiers, attributes and array bounds of declarations is needed than is provided by the concept of compatible types. Two declarations are *token equivalent* if for the two token sequences corresponding to the declarations, after phase 4, there is a sequence of rewrite operations, such that both declarations including array specifications,[168] qualifications and attributes shall consist of the same token sequence, and such that all identifiers that appear are used in the declarations shall be the same and, within their proper context, refer to the same objects, functions, attributes or types.[169] The possible rewrite operations are:

— replacement of digraphs by the token they represent,

— replacement of each multiset of type specifiers by the first equivalent form listed in 6.7.2,

— renaming and eventually adding of parameter names for all parameters that occur in the declaration to the tokens `_Param0`, `_Param1`, . . . , in order,

— addition or removal of white space tokens.

11 **EXAMPLE 1** In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the identifier for that declaration.

```
          [[deprecated]] void f [[deprecated]] (void); // valid
          [[ deprecated ]] void f  [[ deprecated ]]  (void); // valid
```

12 **EXAMPLE 2** Consider the following compatible declarations of a function `sortIt`:

```
/*0*/ void sortIt(size_t nmemb, size_t size, void arr[len][size],
                void* context, int comp(void const[size], void const[size]));
/*1*/ void sortIt(size_t nmemb, size_t size, void arr[len][size],
                void* context, int comp(void const a[size], void const b[size]));
/*2*/ void sortIt(size_t _Param0, size_t _Param1, void _Param2[_Param0][Param1],
                void* _Param3,
                int _Param4(void const _Param5[_Param1], void const _Param6[_Param1]));
/*3*/ void sortIt(size_t nmemb, size_t size, void arr[len][size],
                void* context, int comp(void const[], void const[]));
/*4*/ void sortIt(size_t _Param0, size_t _Param1, void _Param2[_Param0][Param1],
                void* _Param3,
                int (*_Param4)(void const _Param5[_Param1], void const _Param6[_Param1]));
```

Here, the declarations 0 and 1 are token equivalent, because 1 only adds parameter names to the parameters of the callback function `comp`; 2 is also equivalent to these two, since it renames the parameters to a standardized form and otherwise only has some differences in white space.

[168] Thus for token equivalence the rewriting of array parameters to pointers is not applied.

[169] Note that this rewriting is performed on a token level, and that therefore the spelling of types for example through different **`typedef`** matters.

13 In contrast to that, 3 is not token equivalent to any of the previous, because it misses the sizes of the array parameters of the call back. Also, 4 is not equivalent to any of the others because its callback parameter is rewritten to a function pointer.

Forward references: declarators (6.7.8), enumeration specifiers (6.7.2.2), initialization (6.7.12), type names (6.7.9), type qualifiers (6.7.3).

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

```
typedef
extern
static
~~_Thread_local~~ thread_local
auto
~~register~~
```

Constraints

2 ~~At most, one~~

Only the following multisets of storage-class ~~specifier~~ specifiers may be given in the declaration specifiers in ~~a declaration, except that **_Thread_local** may appear with **static** or **extern**~~ the same declaration. Here each line represents a multiset for which the specifiers may appear in any order.

— no storage-class specifier

— **auto**

— **auto extern**

— **auto extern thread_local**

— **auto static**

— **auto static thread_local**

— **auto thread_local**

— **extern**

— **extern thread_local**

— **static**

— **static thread_local**

— **thread_local**

— **typedef** .[170)]

3 In the declaration of an object with block scope, if the declaration specifiers include ~~**_Thread_local**~~ **thread_local** , they shall also include either **static** or **extern**. If ~~**_Thread_local**~~ **thread_local** appears in any declaration of an object, it shall be present in every declaration of that object.

4 ~~**_Thread_local**~~ **thread_local** shall not appear in the declaration specifiers of a function declaration. **auto** shall only appear in the declaration specifiers of a function declaration if it is the declaration part of a function definition or if the corresponding function has already been defined.

170)See "future language directions" (6.11.5).

Semantics

5 The **`typedef`** specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in 6.7.10. The meanings of the various linkages and storage durations were discussed in 6.2.2 and ~~??.~~

~~A declaration of an identifier for an object with storage-class specifier **`register`** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.~~ 6.2.4.

6 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **`extern`**.

7 If an aggregate or union object is declared with a storage-class specifier other than **`typedef`**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

8 If **`auto`** appears with **`extern`**, **`static`** or **`thread_local`**, or if it appears in a declaration at file scope it is ignored for the purpose of determining a storage class or linkage. It then only indicates that the declared type may be inferred from an initializer (for objects see 6.7.13), or from **`return`** statements (for functions see 6.9.1).

9 **NOTE** C++ has abandonned the **`register`** storage class, so programs targetting the C/C++ core should not use this feature and it has been removed from this specification. To obtain similar effects (namely that taking the address of an object is a constraint violation) they should use the **`core`**::**`noalias`** attribute, instead.

Forward references: type definitions (6.7.10), type inference (6.7.13), function definitions (6.9.1).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
~~**`_Bool`**~~
~~**`_Complex`**~~ **`bool`**
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
decltype-specifier
`complex_type`, **`real_type`** or **`generic_type`** specifier macros

Constraints

2 ~~At~~ Unless stated otherwise, at least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each member declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **`void`**

— **`char`**

— **`signed char`**

— `unsigned char`

— `short`, `signed short`, `short int`, or `signed short int`

— `unsigned short`, or `unsigned short int`

— `int`, `signed`, or `signed int`

— `unsigned`, or `unsigned int`

— `long`, `signed long`, `long int`, or `signed long int`

— `unsigned long`, or `unsigned long int`

— `long long`, `signed long long`, `long long int`, or `signed long long int`

— `unsigned long long`, or `unsigned long long int`

— `float`

— `double`

— `long double`

— ~~`_Bool` `float _Complex` `double _Complex` `long double _Complex`~~ `bool`

— atomic type specifier

— ~~struct or union~~ `struct` or `union` specifier

— ~~enum~~ `enum` specifier

— ~~typedef name~~ `typedef` name

— `decltype` specifier

— `complex_type`, `real_type` and `generic_type` specifier macros.

~~The type specifier `_Complex` shall not be used if the implementation does not support complex types (see 6.10.8.2).~~

Semantics

3 Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.10. The characteristics of the other types are discussed in 6.2.5. Declarations for which the type specifiers are inferred from initializers are discussed in 6.7.13.

4 Each of the comma-separated multisets designates the same type~~, except that for bit-fields, it is implementation-defined whether the specifier `int` designates the same type as `signed int` or the same type as `unsigned int`.~~.

5 A declaration that contains no type specifier is said to be *underspecified*. Identifiers that are such declared have incomplete type. Their type can be completed by type inference from an intialization (for objects), from a function call (for lambda parameters) or from `return` statements in a function body (for return types of functions).

6 **NOTE** Note that complex types can be specified as a `decltype` specification or via the `complex_type` macro.

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.10)~~.~~, type inference (6.7.13), predefined macros (6.10.8).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*

  struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**
  struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*

struct-or-union:

  `struct`

union

member-declaration-list:
member-declaration
member-declaration-list member-declaration

member-declaration:
attribute-specifier-sequence$_{opt}$ *specifier-qualifier-list member-declarator-list*$_{opt}$ **;**
static_assert-declaration

specifier-qualifier-list:
type-specifier-qualifier attribute-specifier-sequence$_{opt}$
type-specifier-qualifier specifier-qualifier-list

type-specifier-qualifier:
type-specifier
type-qualifier
alignment-specifier

member-declarator-list:
member-declarator
member-declarator-list **,** *member-declarator*

member-declarator:
declarator
bit-field

bit-field:
declarator$_{opt}$ **:** *constant-expression*

Constraints

2 A member declaration that does not declare an anonymous structure or anonymous union shall contain a member declarator list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 ~~The expression that specifies the width of a~~ A member declarator that is a bit-field shall not appear in a member declarator list, unless the type specifier is **bool**, **signed** or **unsigned**, and there shall be no alignment specifier; the constant expression, the *width* of the bit-field, shall be an integer constant expression ~~with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted~~ M that is not negative, and be less than or equal to **INT_BITFIELD_MAX** (5.2.4.2.1). If the type is **bool**, M shall be 0 or 1. ~~If the~~

~~value is zero, the declaration shall have no declarator~~ If M is 0, there shall be no declarator and no bit of the representation shall be reserved for the bit-field.[171] Otherwise, a bit-field declarator *name*:*M* with qualifier list *CV*, type attributes *TA*, declaration attributes *DA*, and type specifier *T* one of **bool**, **signed** or **unsigned** shall be as if the member *name* had been declared as

~~A bit-field shall have a type that is a qualified or unqualified version of **_Bool**, **signed int**, **unsigned int**,~~

```
[[core::alias]] [[DA]] CV S_M [[TA]] name;
```

where S_M is the type specifier **bool**, **intwidth**(M) or ~~some other implementation-defined type. It is implementation-defined whether atomic types are permitted.~~**uintwidth**(M) (7.20.1.2), respectively, and the constraints corresponding to members with a **core::alias** attribute apply.[172]

5 An attribute specifier sequence shall not appear in a struct-or-union specifier without a member declaration list, except in a declaration of the form:

struct-or-union attribute-specifier-sequence identifier **;**

The attributes in the attribute specifier sequence, if any, are thereafter considered attributes of the **struct** or **union** whenever it is named.

Semantics

6 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

7 Structure and union specifiers have the same form. The keywords **struct** and **union** indicate that the type being specified is, respectively, a structure type or a union type.

8 The optional attribute specifier sequence in a struct-or-union specifier appertains to the structure or union type being declared. The optional attribute specifier sequence in a member declaration appertains to each of the members declared by the member declarator list; it shall not appear if the optional member declarator list is omitted. The optional attribute specifier sequence in a specifier qualifier list appertains to the type denoted by the preceding type specifier qualifiers. The attribute specifier sequence affects the type only for the member declaration or type name it appears in, not other types or declarations involving the same type.

9 The presence of a member declaration list in a struct-or-union specifier declares a new type, within a translation unit. The member declaration list is a sequence of declarations for the members of the structure or union. If the member declaration list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is incomplete until immediately after the } that terminates the list, and complete thereafter.

10 A member of a structure or union may have any complete object type other than a variably modified type.~~In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a ;~~[173]

[171] The only effect of such a member is that it separates a sequence of bit-fields into different packs, see 6.7.15.3.2 for definitions and examples.

[172] Both C and C++ have bit-fields that are "objects" on a scale below a storage unit or that may cross boundaries of storage units. Unfortunately both disagree on their interpretation in terms of types and possible bounds to the number of bits: for example in C an **int** bit-field may be unsigned, or in C++ M is unrestricted but may contain padding. The specification here is a possible intersection between the two languages.

[173] ~~The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.its width is preceded by a colon.~~

A ~~bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.If the value 0 or 1 is stored into a nonzero-width bit-field of type **_Bool**, the value of the bit-field shall compare equal to the value stored; a **_Bool** bit-field has the semantics of a **_Bool**.~~

~~An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order~~ structure or ~~low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.~~

11 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

12 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

13 Within a structure object, the non-bit-field members and ~~the units in which bit-fields reside~~ packs have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member ~~(or if that member is a bit-field, then to the unit in which it resides)~~or *pack* , and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

14 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members ~~(or if a member is a bit-field, then to the unit in which it resides)~~or packs, and vice versa.

15 There may be unnamed padding at the end of a structure or union.

16 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or ~~->~~ $\rightarrow$) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the ~~object~~ storage instance being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

Forward references: the **core** :: **alias** attribute (6.7.15.3.2), exact-width integer types (7.20.1.2).

17 **EXAMPLE 1** The following declarations illustrate the behavior when an attribute is written on a tag declaration:

```
    struct [[deprecated]] S; // valid, [[deprecated]] appertains to struct S
    void f(struct S *s);     // valid, the struct S type has the [[deprecated]]
    struct ⟦deprecated⟧ S; // valid, ⟦deprecated⟧ appertains to struct S
    void f(struct S *s);     // valid, the struct S type has the ⟦deprecated⟧
                             // attribute
    struct S {               // valid, struct S inherits the [[deprecated]] attribute
    struct S {               // valid, struct S inherits the ⟦deprecated⟧ attribute
          int a;             // from the previous declaration
    };
    void g(struct [[deprecated]] S s); // invalid
    void g(struct ⟦deprecated⟧ S s); // invalid
```

18 **EXAMPLE 2** The following illustrates anonymous structures and unions:

```
    struct v {
          union {     // anonymous union
                struct { int i, j; };   // anonymous structure
                struct { long k, l; } w;
          };
          int m;
    } v1;
```

~~A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field~~union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.~~As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.~~

```
v1.i = 2;   // valid
v1.k = 3;   // invalid:  inner structure is not anonymous
v1.w.k = 5; // valid
```

19 **EXAMPLE 3** After the declaration:

```
struct s { int n; double d[]; };
```

the structure `struct s` has a flexible array member `d`. A typical way to use this is:

```
int m = /* some value */;
struct s *p = malloc(sizeof (struct s) + sizeof (double [m]));
```

and assuming that the call to `malloc` succeeds, the object pointed to by `p` behaves, for most purposes, as if `p` had been declared as:

```
struct { int n; double d[m]; } *p;
```

(there are circumstances in which this equivalence is broken; in particular, the offsets of member `d` might not be the same).

20 Following the above declaration:

```
struct s t1 = { 0 };          // valid
struct s t2 = { 1, { 4.2 }};  // invalid
t1.n = 4;                     // valid
t1.d[0] = 4.2;                // might be undefined behavior
```

The initialization of `t2` is invalid (and violates a constraint) because `struct s` is treated as if it did not contain member `d`. The assignment to `t1.d[0]` is probably undefined behavior, but it is possible that

```
sizeof (struct s) >= offsetof(struct s, d) + sizeof (double)
sizeof (struct s) ≥ offsetof(struct s, d) + sizeof (double)
```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

21 After the further declaration:

```
struct ss { int n; };
```

the expressions:

```
sizeof (struct s) >= sizeof (struct ss)
sizeof (struct s) >= offsetof(struct s, d)
sizeof (struct s) ≥ sizeof (struct ss)
sizeof (struct s) ≥ offsetof(struct s, d)
```

are always equal to 1.

22 If `sizeof (double)` is 8, then after the following code is executed:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);
```

and assuming that the calls to `malloc` succeed, the objects pointed to by `s1` and `s2` behave, for most purposes, as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

23 Following the further successful assignments:

```
s1 = malloc(sizeof (struct s) + 10);
```

```
s2 = malloc(sizeof (struct s) + 6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]); // valid
dp = &(s1->d[0]); // valid
*dp = 42;         // valid
dp = &(s2->d[0]); // valid
dp = &(s2->d[0]); // valid
*dp = 42;         // undefined behavior
```

24 The assignment:

```
*s1 = *s2;
```

only copies the member `n`; if any of the array elements are within the first `sizeof (struct s)` bytes of the structure, they might be copied or simply overwritten with indeterminate values.

25 **EXAMPLE 4** Because members of anonymous structures and unions are considered to be members of the containing structure or union, `struct s` in the following example has more than one named member and thus the use of a flexible array member is valid:

```
struct s {
      struct { int i; };
      int a[];
};
```

Forward references: declarators (6.7.8), tags (6.7.2.3).

6.7.2.2 Enumeration specifiers

Syntax

1 *enum-specifier:*

  enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**
  enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **, }**
  enum *identifier*

enumerator-list:

  enumerator
  enumerator-list **,** *enumerator*

enumerator:

  enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$
  enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$ **=** *constant-expression*

Constraints

2 The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **int**.

Semantics

3 The optional attribute specifier sequence in the **enum** specifier appertains to the enumeration; the attributes in that attribute specifier sequence are thereafter considered attributes of the enumeration whenever it is named. The optional attribute specifier sequence in the enumerator appertains to that enumerator.

4 The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.[174] An enumerator with = defines its enumeration constant as the

[174] Thus, the identifiers of enumeration constants declared in the same scope are all required to be distinct from each other

value of the constant expression. If the first enumerator has no =, the value of its enumeration constant is 0. Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

5 Each enumerated type shall be compatible with **char**, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined,[175] but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until immediately after the } that terminates the list of enumerator declarations, and complete thereafter.

6 **EXAMPLE** The following fragment:

```
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
col = claret;
cp = &col;
if (*cp != burgundy)
if (*cp ≠ burgundy)
      /* ... */
```

makes hue the tag of an enumeration, and then declares col as an object that has that type and cp as a pointer to an object that has that type. The enumerated values are in the set $\{0, 1, 20, 21\}$.

Forward references: tags (6.7.2.3).

6.7.2.3 Tags

Constraints

1 A specific type shall have its content defined at most once.

2 Where two declarations that use the same tag declare the same type, they shall both use the same choice of **struct**, **union**, or **enum**.

3 A type specifier of the form

> **enum** *identifier*

without an enumerator list shall only appear after the type it specifies is complete.

4 A type specifier of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

shall not contain an attribute specifier sequence.[176]

Semantics

5 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. Irrespective of whether there is a tag or what other declarations of the type are in the same translation unit, the type is incomplete[177] until immediately after the closing brace of the list defining the content, and complete thereafter.

6 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

7 A type specifier of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**

and from other identifiers declared in ordinary declarators.

[175] An implementation can delay the choice of which integer type until all enumeration constants have been seen.

[176] As specified in 6.7.2.1 above, the type specifier may be followed by a ; or a member declaration list.

[177] An incomplete type can only be used when the size of an object of that type is not needed. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.) The specification has to be complete before such a function is called or defined.

or

> **`enum`** *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **`{`** *enumerator-list* **`}`**

or

> **`enum`** *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **`{`** *enumerator-list* **`,`** **`}`**

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided,[178] the type specifier also declares the identifier to be the tag of that type. The optional attribute specifier sequence appertains to the structure, union, or enumeration type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure, union, or enumeration type whenever it is named.

8 A declaration of the form

> *struct-or-union* *attribute-specifier-sequence*$_{\text{opt}}$ *identifier* **`;`**

specifies a structure or union type and declares the identifier as a tag of that type.[179] The optional attribute specifier sequence appertains to the structure or union type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure or union type whenever it is named.

9 If a type specifier of the form

> *struct-or-union* *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[179]

10 If a type specifier of the form

> *struct-or-union* *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

or

> **`enum`** *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

Recommended practice

11 The fact that tag names are in a different name space than identifiers can lead to portability issues between C and C++, see 6.7.9 below. To avoid incompatibilities, it is recommended to place appropriate **`typedef`** before (for **`struct`** or **`union`**) or after (for **`enum`**) a tag declaration or definition:

```
typedef struct S S;
struct S {
  double data;
  S* next;
};
```

Such a practice ensures that a tag name cannot be reused later as an identifier for a different purpose and that the semantics when seen by a C or C++ translator agree.

12 It is recommended that applications otherwise restrain from using tag names as identifiers whenever possible.

13 **NOTE** C and C++ differ in the ways a tag name may later be used. In particular, a tag name that is not otherwise declared as an identifier can be used as if a **`typedef`** for that name had been declared. Unfortunately this rule could not be extended to C, because there is a major example in POSIX that would conflict with such a mechanism.

[178] If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

[179] A similar construction with **`enum`** does not exist.

14 **EXAMPLE 1** This mechanism allows declaration of a self-referential structure.

```
struct tnode {
      int count;
      struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be an object of the given type and `sp` to be a pointer to an object of the given type. With these declarations, the expression ~~`sp->left`~~ `sp→left` refers to the left `struct tnode` pointer of the object to which `sp` points; the expression ~~`s.right->count`~~ `s.right→count` designates the `count` member of the right `struct tnode` pointed to from `s`.

15 The following alternative formulation uses the `typedef` mechanism:

```
typedef struct tnode TNODE;
struct tnode {
      int count;
      TNODE *left, *right;
};
TNODE s, *sp;
```

16 **EXAMPLE 2** To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

```
struct s1 { struct s2 *s2p; /* ... */ }; // D1
struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to *it*, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the declaration

```
struct s2;
```

can be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

Forward references: declarators (6.7.8), type definitions (6.7.10).

6.7.2.4 Atomic type specifiers

Syntax

1 *atomic-type-specifier:*
  ~~**`_Atomic`**~~ **`atomic_type`** **(** *type-name* **)**

Constraints

2 ~~Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.8.2).~~

The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, an opaque type or a qualified type.

Semantics

3 The atomic type specifier construct is implemented as a mandatory macro (see 6.10.8.1).

4 The properties associated with atomic types are meaningful only for expressions that are lvalues. ~~If the **`_Atomic`** keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.~~

5 **NOTE** C and C++ have no reconcilable syntax for specifying an atomic derivation: C has a keyword **`_Atomic`** that is applied as a specifier (similar to here) and as a qualifier, C++ has a class template `atomic<`*type-name*`>`. Since the C syntax even has ambiguities sticking to the C syntax was not an option. The specification as given here has straight forward implementations in the old syntax for both languages.

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

```
const
```

~~`restrict`~~
~~`volatile`~~
~~`_Atomic`~~ `volatile`

~~**Constraints**~~

~~Types other than pointer types whose referenced type is an object type shall not be restrict-qualified.~~

~~The `_Atomic` qualifier shall not be used if the implementation does not support atomic types (see 6.10.8.2).~~

~~The type modified by the `_Atomic` qualifier shall not be an array type or a function type.~~

Semantics

2 The properties associated with qualified types are meaningful only for expressions that are lvalues.[180)]

3 If the same qualifier appears more than once in the same specifier-qualifier list or as declaration specifiers, either directly or via one or more **`typedef`**s, the behavior is the same as if it appeared only once. ~~If other qualifiers appear along with the `_Atomic` qualifier the resulting type is the so-qualified atomic type.~~

4 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.[181)]

5 An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[182)] What constitutes an access to an object that has volatile-qualified type is implementation-defined.

6 ~~An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in ?? below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.The intended use of the `restrict` qualifier (like the `register` storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).~~

~~If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.~~

For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

7 **NOTE** C also has the `restrict` and `_Atomic` qualifiers. These have never been integrated to C++, so they should not be

---

180)The implementation can place a **`const`** object that is not **`volatile`** in a read-only storage instance. Moreover, a storage instance for such an object need not be addressable if its address is never used.

181)This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

182)A **`volatile`** declaration can be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared are not allowed to be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

used by applications that target the C/C++ core. For the first, this specification proposes the use of the `core::noalias` attribute in the form when it is applied to pointer declarators. To avoid certain ambiguities, the possible syntax is a bit more restricted than the use as a qualifier. Instead of an `_Atomic` qualification, an `atomic_type` specification may be used.

8 **EXAMPLE 1** An object declared

```
extern const volatile int real_time_clock;
```

might be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

9 **EXAMPLE 2** The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;      // valid
cs = ncs;      // violates modifiable lvalue constraint for =
pi = &ncs.mem; // valid
pi = &cs.mem;  // violates type constraints for =
pci = &cs.mem; // valid
pi = a[0];     // invalid:  a[0] has type “const int *”
```

~~The declaration specifies that `p` has the type “pointer to volatile atomic int”, a pointer to a volatile-qualified atomic type.~~

6.7.4 The `constexpr` specifier

Constraints

1 ~~Let `D` be~~ If the **`constexpr`** specifier appears in a declaration, it shall appear in all declarations of the same object or function; it shall only appear in a declaration in file or block scope, and it shall not appear together with a storage class specifier other than **`auto`**. If the identifier has linkage and if the **`inline`** specifier is applied as well, the linkage is external; otherwise it is internal.

2 If the **`constexpr`** specifier appears in a declaration of an ~~ordinary identifier that provides a means of designating an object `P` as a restrict-qualified pointer to type `T`.~~ object, the object has static storage duration, it shall not be **`volatile`** qualified, a **`const`** qualification shall be implied and the declaration serves as a definition. All lvalue conversions of such an object shall result in the initializer value as determined during translation. If the object has external linkage the definition is an inline definition and if in the whole program no other access than an lvalue conversion is made to the object, no external definition is needed. The constraints for the initialization of objects with static storage duration apply, with the additional constraints that none of the following constructs shall occur:

— an implicit or explicit pointer-to-pointer or array-to-pointer conversion,

— an implicit initialization of a pointer object or member other than by a null pointer,

— an explicit initialization of a pointer object or member other than by a null pointer constant or by the result of an integer-to-pointer cast.[183]

~~If `D` appears inside a block and does not have storage class **`extern`**, let `B` denote the block.If `D` appears in the list of parameter declarations~~ If the **`constexpr`** specifier appears in a declaration of a function~~definition, let `B` denote the associated block. Otherwise, let `B` denote the block of~~

[183]Thus, if the object has pointer type or is an agregate or union type with a pointer element or member, an explicit initializer for it (or the element or member) shall only be a null pointer constant or a pointer value that is formed by a cast from an integer constant expression, and no indirect initialization through another member shall result in a value that is different from null.

~~(or the block of whatever function is called at program startup in a freestanding environment)~~, the **core::concurrent** attribute is implied and the corresponding constraints apply. For function declarations and lambda expressions that have the **constexpr** specifier and that have a pointer return type with no **core::alias** attribute that links it to one of its parameters, the **core::noalias** attribute is implied and the corresponding constraints apply.[184] For a lambda expresssion, the **constexpr** specifier is implied if it has the **core::concurrent** attribute and the above requirements on a possible pointer return value apply.

4 ~~In what follows, a pointer expression E is said to be *based* on object P if (at some sequence point in the execution of B prior to the evaluation of E) modifying P to point to a copy of the array object into which it formerly pointed would change the value of E.~~ A function call of a function designator or lambda value with a **constexpr** specifier can be used in a context that requires a constant expression if the following contraints are fulfilled.

— If it is a function, the function definition shall be visible or the function shall belong to an implementation-defined set of functions that are usable for this purpose. This set of functions shall contain at least those C library functions that are specified with **constexpr**.

— All arguments to the call shall be constant expressions that are not address constants.

— If the return type is a pointer type, the return value shall be a null pointer or the result of an integer-to-pointer cast of an integer constant expression.

— In addition to the parameters, captures, and locally defined or allocated objects, the execution shall only access objects that are declared with **constexpr**, that are otherwise permitted in a constant expression or those identifiers in the **core::evaluates** or **core::modifies** attributes that have thread local storage duration.

— Pointer values shall only refer to objects that do not have static storage duration; ~~Note that "based" is defined only for expressions with pointer types.~~

~~During each execution of B, let L be any lvalue that has &L based on P. If L is used to access the value of the object X that it designates, and X is also modified (by anymeans) , then the following requirements apply: T shall not be const-qualified. Every other lvalue used to access the valueof X shall also have its address based on P. Every access that modifies X shall be considered also to modify P, for~~ when they are subject to the ~~purposes of this subclause. If P is assigned the~~ unary $*$ operator, the provenance of the referred object shall be visible and the effect shall be as if the object definition (if it has any) had directly been used for the lvalue expression; if such an object has no definition, a definition with its effective type is assumed.

— No lvalue conversion shall be formed for an object with an indeterminate value.

— The execution of the function or lambda with the specified parameters shall only exhibit constructs that have defined behavior for the specified values.

— No operation shall be formed for which the result is unspecified.

— No implementation-defined limits as specified in 5.2.4 shall be exceeded.

— The the **errno** and **fenv** channels shall not be modified.

— For any function call the arguments and captured objects that are not pointer values are considered as if they were constant expressions. The present constraints for such function calls are then recursively applied were possible. Additionally:

- Pointer values as above may be arguments or captured variables of such **constexpr** function calls.

[184] The two aliasing attributes are not exclusive; if both are given this indicates that a function or lambda conditionally returns the pointer to a newly allocated storage instance or one of its parameter values.

- The storage management functions from `<stdlib.h>` (7.22.4) may be called, provided that all allocations are deallocated before the end of the call. An implementation-defined limit may be imposed on the maximum size and number of allocations that can be effected during such a call.
- If a pointer value that is not a null pointer or converted integer constant expression is the return value of such a call it shall be the return value of an allocation function or the value of ~~a pointer expression `E` that is based on another restricted pointer object `P2`, associated with block `B2`, then either the execution of `B2` shall begin before the execution of `B`, or the execution of `B2` shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.~~ one of the arguments to the particular call. In the first case, the called function shall have a **`core::noalias`** attribute; in the second it shall have a **`core::alias`** attribute with the corresponding parameter name.

5 ~~Here an execution of `B` means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with `B`.~~ If a function call of a function designator or lambda value with a **`constexpr`** specifier fulfills the above constraints it shall be considered a constant expression; if does not and the function call is valid in the context where it appears, the function call is effected in that context whenever it is met during execution and a different set of constraints may apply in consequence.[185]

Semantics

6 ~~A translator is free to ignore any or all aliasing implications of uses of **`restrict`**. The file scope declarations assert that if an object is accessed using one of `a`, `b`, or `c`, and that object is modified anywhere in the program, then it is never accessed using either of the other two. The function parameter declarations in the following example assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other. The translator can make this no-aliasing inference based on the parameter declarations alone, without analyzing the function body.~~ The **`constexpr`** specifier indicates that the identifier or lambda value may be used in a context where a constant expression is required. If a function designator or lambda is used in such a context the contraints ensure that a call that would be erroneous during program execution can be identified during translation and that a diagnostic is issued.

7 ~~The benefit of the **`restrict`** qualifiers is that they enable a translator to make an effective dependence analysis of function `f` without examining any of the calls of `f` in the program~~ If such a call would be successful during program execution, it is executed during translation as-if by a separate thread of execution. During such an execution all full expressions are evaluated in sequence as the control flow implies and the result of each such an evaluation is considered a constant expression. The determined return value is used as a constant of the corresponding type in the calling context.[186]

Recommended practice

8 If a function designator or lambda value is used in a context that requires a constant expression and the contraints are violated it is recommended that the translator does not produce an executable program image.

9 **NOTE 1** The **`constexpr`** specifier heavily relies on the properties of the **`core::concurrent`** attribute and the other attributes that it implies, namely the **`core::stateless`**, **`core::noleak`**, **`core::state_invariant`**, and **`core::state_conserving`** attributes. In all, these imply that no pointer value is leaked, exposed or synthesized by a function call, that the only state dependencies come from well identified input channels, and that a complete data flow and aliasing analysis can be performed at compile time. The requirements for the use within constant expressions then further narrow the field to dependencies from other values that are already known to be constant expressions, and whose properties do not depend on linkage but on translation only. ~~The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example,~~

[185] So in particular a call that serves to determine an array length is considered, if possible, to be a constant expression and the array then is not a VLA. On the other hand, if it does not fulfill the constraints but still is valid, the array is a VLA and the corresponding constraints, for example concerning initialization, apply.

[186] This means in particular that during such an execution no variably modified type (VM) will be formed.

10 **NOTE 2** The result of a call to a **constexpr** function or lambda in the context of a constant expression is fully independent of general or specific properties of the address space. In particular, no pointer-to-integer casts and no pointer comparisons, even equality, between pointers with different provenance may be performed. The feature allows the dependency of the computation from some tread local state, but since it is executed as-if taking place in a separate thread, changing this state has no impact on other calls within the evaluation of constant expressions. Thus, such changes to thread local state are not observable by any execution and the only effect of such a call is its return value, which then can be recorded at translation time.

11 **NOTE 3** The constraints for calls within constant expressions enforce that all possible undefined or unspecified behavior can be detected at compile time. This includes out-of-bound access of arrays, lvalue conversion of uninitialized variables, arithmetic overflow, domain errors, null pointer dereference, comparison of unrelated pointer values, leaks and the ~~second call of `f` in `g` has undefined behavior because each of `d[1]` through `d[49]` is accessed through both `p` and `q`.~~ unsequenced modification of objects.

12 **EXAMPLE** As examples for **constexpr** objects and functions that are evaluated during translation consider the following:

```
        void g(void)
        {
                extern int d[100];
                f(50, d + 50, d); // valid
                f(50, d +  1, d); // undefined behavior
struct pair { double val[2]; };
constexpr struct pair diff(struct pair A, struct pair B) {
  struct pair ret = {
    .val = {
      [0] = B.val[0] - A.val[0],
      [1] = B.val[1] - A.val[1],
    },
  };
  return ret;
}
```

~~The function parameter declarations illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if `a` and `b` are disjoint arrays,~~ First, observe that δ is a ~~call of the form `h(100, a, b, b)` has defined behavior, because array `b` is not modified within function `h`. The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only "outer-to-inner" assignments between restricted pointers declared in nested blocks have defined behavior.~~ named constant of type **double**, and that, although **double** is a basic type, such a constant cannot otherwise be formed other than by the equivalent **inline static const** specifiers and qualification, see below. The objects `a`, `b`, α and β all are constants of type **struct** `pair`; the initializers are valid because **sin** and **asin** are C library functions with **constexpr** and the argument values are within the valid ranges. In contrast to that, the argument for the initializer of ψ is outside the valid range of **asin**, and therefore an error must be diagnosed.

~~The one exception allows the value of a restricted pointer to be carried out of the block in which it~~
13 ~~(or~~ The initializers for the objects `c` and γ use the **constexpr** functions `inter` and, ~~more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`. Suppose that a programmer knows that references of the form `p[i]` and `q[j]` are never aliases in the body of a function: There are several ways that this information could be conveyed to a translator using the **restrict** qualifier. Example 2 shows the most effective way, qualifying all pointer parameters, and can be used provided that neither `p` nor `q` becomes based on the other in the function body. A potentially effective alternative is: Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use subtler reasoning: that the const-qualification of `q` precludes it becoming based on `p`. There is also a requirement that`q` is not modified, so this alternative cannot be used for the function in Example 2, as written. Another potentially effective alternative is: Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use even subtler reasoning: that this combination of **restrict** and**const** means that objects referenced using `q` cannot be modified, and so no modified object can be referenced using both `p` and `q`. The least effective alternative is : Here~~

~~the translator can make the no-aliasing inference only by analyzing the body of the function and proving that q cannot become based on p. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator is required to assume that aliases are present because assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and thus need to conservatively assume that aliases are present~~indirectly, `diff`. Both are valid, because the arguments are valid and no arithmetic exception occurs. The initializer of χ is invalid because the `val[0]` element for both parameters are the same, and thus their difference is zero. The computation for `slope` then has a division by zero error, and therefore the whole value computation is invalid and must be diagnosed.

14 Similar reason as above show that the initialization of π is a valid constant expression of type **`long double`**. The call **`strlen`**`(`π`, 16)` is then a valid integer constant expression that provides the length of the textual representation of π in hexadecimal, and thus this expression can be used for the declaration of the type `strType`. Now the lambda expression only uses the **`totext`** type-generic macro, which for the given arguments has the **`constexpr`** specifier, and behaves therefore as if it were itself specified with **`constexpr`**. As a consequence the lambda can be used to initialize π`Str`, and the address of the `string` member can be used to initialize a **`static`** variable.

Forward references:

~~**Syntax**~~

~~*function-specifier:*~~
~~**`inline`**~~
~~**`_Noreturn`**~~

the **`inline`** specifier (6.7.5), the **`core`**`::`**`concurrent`** attribute (6.7.15.4.13), environmental limits (5.2.4), errors `<errno.h>` (7.5), floating-point environment `<fenv.h>` (7.6), mathematics `<math.h>` (7.12), the **`totext`** type-generic macro (7.22.2.1), the **`strlen`** type-generic macro (7.24.6.4).

6.7.5 The `inline` specifier

Constraints

1 ~~Function specifiers shall be used only in the~~ An **`inline`** specifier shall only appear in a declaration at file scope. If the **`inline`** specifier appears in any declaration of an identifier ~~for a function~~with external linkage, it shall also appear in the file scope declaration that is met first.

2 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static or thread storage duration, and shall not contain a reference to an identifier with internal linkage.

3 In a hosted environment, no ~~function specifier (s)~~ **`inline`** specifier shall appear in a declaration of **`main`**.

4 If the declaration declares an object that is **`const`** but not **`volatile`** qualified, a **`constexpr`** specifier is implied and all constraints apply.

Semantics

5 ~~A function~~ An **`inline`** specifier may appear more than once; the behavior is the same as if it appeared only once. If in any translation unit an identifier with external linkage is declared inline, it shall be declared inline in any of them.

6 A function declared with an **`inline`** ~~function~~ specifier is an *inline function*. Making a function an inline function suggests that calls to the function be as fast as possible.[187] The extent to which such suggestions are effective is implementation-defined.[188]

[187]By using, for example, an alternative to the usual function call mechanism, such as "inline substitution". Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

[188]For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **`inline`** declaration.

7 Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an **`inline`** function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the **`inline`** function specifier without **`extern`**, then the definition in that translation unit is an *inline definition*. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.[189)]

8 ~~A function declared with a **`_Noreturn`** function specifier shall not return to its caller.~~ All inline and the external definition of a function, if any, shall behave the same such that an observation of the return value and of side effects would not be able to distinguish between them.[190)] It is unspecified if an object definition of a **`const`**-qualified type and of static storage duration within the body of an inline function refers to a single object or to distinct objects for each of the definitions of the function.

9 An object declared with an **`inline`** specifier is an *inline object*; any object with internal linkage can be an inline object. For an object with external linkage the following restrictions apply: If a file scope declaration for such an object in a translation unit includes the **`inline`** specifier without **`extern`**, then this declaration shall be a definition with an initializer; if no other declaration occurs in file scope, this is an *inline definition*. An inline definition does not provide an external definition for the object, and does not forbid an external definition in another translation unit.

10 All initializers of an inline object shall not evaluate the object and shall evaluate to the same constant value. All evaluations that refer to an inline object other than by an lvalue conversion shall refer to the external definition.[191)]

Recommended practice

11 ~~The implementation should produce a diagnostic message for a function declared with a **`_Noreturn`** function specifier that appears to be capable of returning to its caller.~~ Because inline definitions in separate translation units constitute different definitions, and because the one possible external definition may add yet another definition, the **`static`** declaration of a **`const`**-qualified object within an inline function, may effectively refer to several objects with the same content. This is for example the case for the **`__func__`** predefined identifier, that, for the same inline function, may have several representations. This is in strong contrast to C++ where it is guaranteed that such objects are only represented once.

12 It is recommended that applications that target the common C/C++ core do not make assumptions about the representation of these objects and that they are made robust for the possibility that such an object has one or several representations. Applications that need to ensure that a unique address is used should move the definition of the object to file scope and make it inline.

13 When possible, it is recommended that implementations diagnose the usage of the address of such an object that could result in a difference in behavior between implementations that represent one or several objects. In particular, it is recommended to diagnose the escape of such an address from an inline function.

14 **NOTE 1** C and C++ differ slightly in their handling of inline functions. Whereas C enforces the use of an external definition in certain situations, in particular if the address of an inline function is used other than in a function call, C++ always guarantees that an external definition (called an instantiation) is emitted if there is need for it. This choice for C is deliberate, because traditionally C is often used in contexts that have severe constraints on the memory size for the program image. So a systematic generation of unused function definitions in all translation units is avoided.

---

189) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

190) That means, no observable difference in side effects such as for the change of the value of an objects of static or thread-local storage shall occur, regardless if an inline or extern definition is choosen for a particular execution and function call. Nevertheless, differences in access to outside resources such as a clock, an input/output device, or scheduling resources of the underlying operation system may occur if the execution times of different definitions differ.

191) This includes the case that an inline object, **`const`**-qualified or not, appears as operand of the unary & operator.

15 **NOTE 2** This specification follows C++ (and extends C) by requiring that the effective semantics of inline and external definitions have to agree. It follows C (and extends C++) by requiring that no non-**const** qualified objects with internal linkage may be accessed by inline functions.

16 **NOTE 3** C currently has no inline objects, so this specification imposes an extension of the C language. The definitions presented here not only serve the purpose of programming invariantly in C and C++, but also to provide a tool to specify compile time constants of any object type.

17 **NOTE 4** For both, functions and objects, the choice has been made to follow mostly the C model for instantiation, that is, to require that an external definition must be presented explicitly for functions or objects that use the address or that form a modifiable lvalue. So this part of the specification extends the C++ language by imposing more constraints on well-formed programs. The special case for inline constants, see 6.6, allows to avoid the need for instantiations, if the address of the object is never used.

18 **EXAMPLE 1** The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit.

```
inline double fahr(double t)
{
        return (9.0 * t) / 5.0 + 32.0;
        return (9.0 × t) / 5.0 + 32.0;
}

inline double cels(double t)
{
        return (5.0 * (t - 32.0)) / 9.0;
        return (5.0 × (t - 32.0)) / 9.0;
}

extern double fahr(double);    // creates an external definition
extern double fahr(double);    // forces the definition to be external

double convert(int is_fahr, double temp)
{
        /* A translator may perform inline substitutions */
        return is_fahr ? cels(temp): fahr(temp);
}
```

19 Note that the definition of `fahr` is an external definition because `fahr` is also declared with **extern**, but the definition of `cels` is an inline definition. Because `cels` has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either can be used for the call.

20 **EXAMPLE 2** The declaration of an inline object with external linkage may or may not result in an external definition. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit.

```
_Noreturn void f () {
        abort(); // ok
inline const void*const self = &self;     // invalid

inline const size_t aware = sizeof aware; // valid, not evaluated

inline const double π = 3.14159265358979323846;

inline const double π^2 = π × π;

extern const double π^2; // forces the definition to be external

static const double π^0;

const double*const power[3] = { [0] = &π^0, [1] = &π, [2] = &π^2, };

const double* g(void) {
        return &π;
}
```

```
    _Noreturn void g (int i) { // causes undefined behavior if i <= 0
        if (i > 0) abort();
    double f(void) {
        return π × *g();
    }
```

21 The inline definition of `self` is invalid because the identifier `self` is evaluated and because the initializer expression is neither a null pointer constant nor a converted integer constant expression. If the **`inline`** specifier were omitted, the definition would be valid, but usually the concrete address to which the object would be initialized only manifests when several translation units are linked to form the final program image. Thus that address can not be used without knowing the external definition.

22 For `aware` such restrictions do not apply because the type information is present at the point of initialization in any translation unit.

23 Note that the definition of π^2 is an external definition because it is also declared with **`extern`**, but that the definition of π is an inline definition. So in the definition of `power`, π^0 and π^2 refer to objects in the same translation unit, but π has to refer to an external definition of another one (see 6.9).

24 Within the body of function `f`, the evaluation of π uses the constant value of the initializer. The evaluation of `g()` returns the address of the external definition of π, so `*g()` forms an lvalue conversion of an external definition in another translation unit. Whether or not this results in a memory load operation of that external definition or a usage of the constant, is unspecified.

Forward references: function definitions (6.9.1).

6.7.6 Alignment specifier

Syntax

1 *alignment-specifier:*

~~**`_Alignas`**~~ **`alignas`** **`(`** *type-name* **`)`**
~~**`_Alignas`**~~ **`alignas`** **`(`** *constant-expression* **`)`**

Constraints

2 An alignment specifier shall appear only in the declaration specifiers of a declaration, or in the *specifier-qualifier* list of a member declaration, or in the type name of a compound literal. An alignment specifier shall not be used in conjunction with ~~either of~~ the storage-class specifiers **`typedef`** ~~or **`register`**~~, nor in a declaration of a function or ~~bit-field~~if a **`core`**::**`noalias`** attribute is applied, including bit-fields.

3 The constant expression shall be an integer constant expression. It shall evaluate to a valid fundamental alignment, or to a valid extended alignment supported by the implementation for an object of the storage duration (if any) being declared, or to zero.

4 An object shall not be declared with an over-aligned type with an extended alignment requirement not supported by the implementation for an object of that storage duration.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the type of the object or member being declared.

Semantics

6 The first form is equivalent to ~~**`_Alignas(_Alignof(`**~~**`alignas(alignof(`***type-name*`))`.

7 The alignment requirement of the declared object or member is taken to be the specified alignment. An alignment specification of zero has no effect.[192] When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

8 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment

[192] An alignment specification of zero also does not affect other alignment specifications in the same declaration.

specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.7 The `_Noreturn` specifier

Constraints

1 The **`_Noreturn`** specifier shall be used only in the declaration of an identifier for a function.

2 In a hosted environment, no **`_Noreturn`** specifier shall appear in a declaration of **`main`**.

Semantics

3 The **`_Noreturn`** specifier may appear more than once; the behavior is the same as if it appeared only once.

4 A function declared with a **`_Noreturn`** specifier shall not return to its caller.

Recommended practice

5 The implementation should produce a diagnostic message for a function declared with a **`_Noreturn`** specifier that appears to be capable of returning to its caller.

6 **EXAMPLE**

```
_Noreturn void f () {
        abort(); // ok
}

_Noreturn void g (int i) { // causes undefined behavior if i ≤ 0
        if (i > 0) abort();
}
```

Forward references: function definitions (6.9.1).

6.7.8 Declarators

Syntax

1 *declarator:*

pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator:

identifier attribute-specifier-sequence$_{\text{opt}}$
(*declarator* **)**
array-declarator attribute-specifier-sequence$_{\text{opt}}$
function-declarator attribute-specifier-sequence$_{\text{opt}}$

~~*array-declarator:*~~
~~*direct-declarator* **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**~~
~~*direct-declarator* **[** **static** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**~~
~~*direct-declarator* **[** *type-qualifier-list* **static** *assignment-expression* **]**~~
~~*direct-declarator* **[** *type-qualifier-list*$_{\text{opt}}$ ***** **]**~~
~~*function-declarator:*~~
~~*direct-declarator* **(** *parameter-type-list*$_{\text{opt}}$ **)**~~
~~*pointer:*~~
~~***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$~~
~~***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *pointer type-qualifier-list:*~~
~~*type-qualifier*~~
~~*type-qualifier-list type-qualifier parameter-type-list:*~~
~~*parameter-list*~~
~~*parameter-list* **, ...** *parameter-list:*~~
~~*parameter-declaration*~~
~~*parameter-list* **,** *parameter-declaration parameter-declaration:*~~
~~*attribute-specifier-sequence*$_{\text{opt}}$ *declaration-specifiers declarator*~~

~~*attribute-specifier-sequence*$_{opt}$ *declaration-specifiers abstract-declarator*$_{opt}$~~

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

3 A *full declarator* is a declarator that is not part of another declarator. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified.

4 In the following subclauses, consider a declaration

```
T D1
```

where `T` contains the declaration specifiers that specify a type *T* (such as **`int`**) and `D1` is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

5 If, in the declaration "`T D1`", `D1` has the form

identifier attribute-specifier-sequence$_{opt}$

then the type specified for *ident* is *T* and the optional attribute specifier sequence appertains to ~~`D1`~~ the entity as it is declared.

6 If, in the declaration "`T D1`", `D1` has the form

```
( D )
```

then *ident* has the type specified by the declaration "`T D`". Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

7 There is no syntax derivation to form declarators of lambda types. Values of lambda type can be formed by lambda expressions and their type can be inferred.

Implementation limits

8 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or **`void`** type, either directly or via one or more **`typedef`** s.

Forward references: array declarators (6.7.8.2), type definitions (6.7.10)~~.~~, type inference (6.7.13).

6.7.8.1 Pointer declarators

Syntax

pointer:

        `*` *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$

        `*` *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*

Semantics

1 If, in the declaration "`T D1`", `D1` has the form

`*` *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$ `D`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*". For each type qualifier in the list, *ident* is a so-qualified pointer. The optional attribute specifier sequence appertains to the pointer and not the object pointed to.

2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

3 **EXAMPLE** The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value".

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by `ptr_to_constant` cannot be modified through that pointer, but `ptr_to_constant` itself can be changed to point to another object. Similarly, the contents of the **int** pointed to by `constant_ptr` can be modified, but `constant_ptr` itself always points to the same location.

4 The declaration of the constant pointer `constant_ptr` can be clarified by including a definition for the type "pointer to **int**".

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares `constant_ptr` as an object that has type "const-qualified pointer to **int**".

6.7.8.2 Array declarators

Syntax

1 *array-declarator:*
 direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
 direct-declarator **[** **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
 direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**

Constraints

2 In addition to optional type qualifiers and the keyword **static**, the [and] may delimit an ~~expression or *~~ assignment expression. If they delimit an ~~expression (which specifies the size of an array), the expression~~ assignment expression, it shall have an integer type. If the assignment expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

3 If an identifier is declared as having a variably modified type, it shall be an ordinary identifier (as defined in 6.2.3), have no linkage, and have either block scope or function prototype scope~~.~~; if the implementation defines the macro **__CORE_NO_VLA__** a definition that has a variable length array type shall have function prototype scope.[193] If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

4 If two declarations of the same array type are visible, both shall have compatible element types, and if both assignment expressions are present, and are integer constant expressions, then both shall have the same constant value.

Semantics

5 If, in the declaration "`T D1`", `D1` has one of the forms:

`D [` *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]** *attribute-specifier-sequence*$_{opt}$
`D [` **static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]** *attribute-specifier-sequence*$_{opt}$
`D [` *type-qualifier-list* **static** *assignment-expression* **]** *attribute-specifier-sequence*$_{opt}$
~~D [*type-qualifier-list*$_{opt}$ *] *attribute-specifier-sequence*$_{opt}$~~

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[194] The optional attribute specifier sequence appertains to the array. (See 6.7.8.3 for the meaning of the optional type qualifiers and the keyword **static**.)

6 ~~If the size is not present, the arraytype is an incomplete type.~~ The value of the assignment expression is the *size* of the array. If the size is ~~* instead of being an expression~~not present, the

[193] A parameter definition as a VLA, is, as all array parameters, rewritten to a pointer type.

[194] When several "array of" specifications are adjacent, a multidimensional array is declared.

array type is ~~a type of unspecified size, which can only be used in declarations or typenames with function prototype scope;such arrays are nonetheless complete types~~an incomplete type. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. ~~(Variable length arrays are a conditional feature that implementations need not support; see 6.10.8.2.)~~

7 If the size is an expression that is not an integer constant expression: if it occurs in a declaration ~~at function prototype scope~~that is not a definition, it is ~~treated as if it were replaced by *~~ not evaluated; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of an ~~**_Alignof**~~ **alignof** operator, that expression is not evaluated.

8 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both ~~size~~ specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, ~~it is undefined behavior if the two size specifiers evaluate to unequal values.~~ the two specifiers shall evaluate to the same size.

9 **NOTE** Traditionally, C and C++ differ in some of the aspects of array declarations. C has VM types since C99, but made them optional with a feature macro __STDC_NO_VLA__ in C11. This possibility not withstanding, there is no known implementation that would conform to C17 that defines that feature macro. C++ has no VM types. VM types, with the possibility to forbid definitions of VLA in block scope are nevertheless proposed for this core specification, because they provide a convenient tool to enforce propagation of array sizes. In particular such an enforcement is possible from the caller of a function with array parameters into the function body, without changing function ABIs and without jeopardizing performance or safety.

10 **EXAMPLE 1**

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

11 **EXAMPLE 2** Note the distinction between the declarations

```
extern int *x;
extern int y[];
```

The first declares x to be a pointer to **int**; the second declares y to be an array of **int** of unspecified size (an incomplete type), the storage instance for which is defined elsewhere.

12 **EXAMPLE 3** The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;     // invalid: not compatible because 4 != 6
      p = a;     // invalid: not compatible because 4 ≠ 6
      r = c;     // compatible, but defined behavior only if
                 // n == 6 and m == n+1
                 // n ≡ 6 and m ≡ n+1
}
```

13 **EXAMPLE 4** All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the ~~**_Thread_local**~~ **thread_local**, **static**, or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type), unless the implementation define __CORE_NO_VLA__. Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
    extern int n;
    int A[n];                      // invalid:  file scope VLA
    extern int (*p2)[n];           // invalid:  file scope VM
    int B[100];                    // valid:  file scope but not VM

    void fvla(int m, int C[m][m]);      // valid:  VLA with prototype scope

    void fvla(int m, int C[m][m])       // valid:  adjusted to auto pointer to VLA
    {
          typedef int VLA[m][m];        // valid:  block scope typedef VLA

          struct tag {
                int (*y)[n];            // invalid:  y not ordinary identifier
                int z[n];               // invalid:  z not ordinary identifier
          };
    #ifndef __CORE_NO_VLA__
          int D[m];                     // valid:  auto VLA
          int G[m] = {};                // invalid:  attempt to initialize VLA
    #endif
          static int E[m];              // invalid:  static block scope VLA
          extern int F[m];              // invalid:  F has linkage and is VLA
          int (*s)[m];                  // valid:  auto pointer to VLA
          extern int (*r)[m];           // invalid:  r has linkage and points to VLA
          static int (*q)[m] = &B;      // valid:  q is a static block pointer to VLA
    }
```

Forward references: function declarators (6.7.8.3), function definitions (6.9.1), initialization (6.7.12).

6.7.8.3 Function declarators

Syntax

function-declarator:
 direct-declarator **(** *parameter-type-list*$_{opt}$ **)**

type-qualifier-list:
 type-qualifier
 type-qualifier-list type-qualifier

parameter-type-list:
 parameter-list
 parameter-list **, ...**

parameter-list:
 parameter-declaration
 parameter-list **,** *parameter-declaration*

parameter-declaration:
 attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator*
 attribute-specifier-sequence$_{opt}$ *declaration-specifiers abstract-declarator*$_{opt}$

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is ~~`register`~~ `auto`.

3 A parameter declaration without type specifier shall not be formed, unless it includes a storage class specifier and unless it appears in the parameter list of a lambda expression.

4 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

Semantics

5 If, in the declaration "`T D1`", `D1` has the form

`D (` *parameter-type-list*$_{opt}$ `)` *attribute-specifier-sequence*$_{opt}$
and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*". The optional attribute specifier sequence appertains to the function type.

6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

7 ~~A~~ After the declared types of all parameters have been determined in order of declaration, any declaration of a parameter as "array of *type*" shall be adjusted to "qualified or unqualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

8 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

9 If the list terminates with an ellipsis (~~`, ...`~~`, ...`), no information about the number or types of the parameters after the comma is supplied.[195]

10 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

11 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

12 If the function declarator is not part of a definition of that function, parameters may have incomplete type~~and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types~~.

13 The storage class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. The optional attribute specifier sequence in a parameter declaration appertains to the parameter.

14 For a function declarator without a parameter type list: if it is part of a definition of that function the function has no parameters and the effect is as if it were declared with a parameter type list consisting of the keyword **`void`**; otherwise it specifies that no information about the number or types of the parameters is supplied.[196] A function declarator provides a *prototype* for the function if it includes a parameter type list.[197] Otherwise, a function declaration is said to have no prototype.

15 For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type has none and is not part of a function definition, the parameter list shall not have an ellipsis terminator. In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

16 **NOTE** C and C++ have different rules for the visibilty of function parameters: for C a parameter is visible starting at the end of its declaration, whereas for C++ it is only visible starting in the function body, if the declaration also happens to be a definition. This specification opted for the C variant, because this rule implies that one parameter can be used for the declaration of the type of another. That possibility is important wherever there is a need to ensure consistency between types or array lengths.

17 **EXAMPLE 1** The declaration

```
int f(void), *fip(), (*pfi)();
```

[195] The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.
[196] See "future language directions" (6.11.6).
[197] This implies that a function definition without a parameter list provides a prototype, and that subsequent calls to that function in the same translation unit are constrained not to provide any argument to the function call. Thus a definition of a function without parameter list and one that has such a list consisting of the keyword **`void`** are fully equivalent.

declares a function `f` with no parameters returning an `int`, a function `fip` with no parameter specification returning a pointer to an `int`, and a pointer `pfi` to a function with no parameter specification returning an `int`. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the pointer result to yield an `int`. In the declarator `(*pfi)()`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an `int`.

18 If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions `f` and `fip` have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer `pfi` has block scope and no linkage.

19 **EXAMPLE 2** The declaration

```
int (*apfi[3])(int *x, int *y);
```

declares an array `apfi` of three pointers to functions returning `int`. Each of these functions has two parameters that are pointers to `int`. The identifiers `x` and `y` are declared for descriptive purposes only and go out of scope at the end of the declaration of `apfi`.

20 **EXAMPLE 3** The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function `fpfi` that returns a pointer to a function returning an `int`. The function `fpfi` has two parameters: a pointer to a function returning an `int` (with one parameter of type `long int`), and an `int`. The pointer returned by `fpfi` points to a function that has one `int` parameter and accepts zero or more additional arguments of any type.

21 **EXAMPLE 4** The following prototype has a variably modified parameter.

```
void addscalar(int n, int m,
      double a[n][n*m+300], double x);

int main()
{
      double b[4][308];
      addscalar(4, 2, b, 2.17);
      return 0;
}

void addscalar(int n, int m,
      double a[n][n*m+300], double x)
{
      for (int i = 0; i  <  n; i++)
            for (int j = 0, k = n*m+300; j  <  k; j++)
                  // a is a pointer to a VLA with n*m+300 elements
                  a[i][j] += x;
}
```

22 **EXAMPLE 5** The following are ~~all~~ compatible function prototype declarators.

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
void f(double (* a)[5]);
void f(double a[][5]);
void f(double a[3][5]);
void f(double a[static 3][5]);
```

(Note that the last declaration also specifies that the argument corresponding to `a` in any call to `f` can be expected to be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Forward references: function definitions (6.9.1), type names (6.7.9).

6.7.9 Type names

Syntax

1 *type-name:*
    specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
    pointer
    pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
    (*abstract-declarator* **)**
    array-abstract-declarator attribute-specifier-sequence$_{opt}$
    function-abstract-declarator attribute-specifier-sequence$_{opt}$

array-abstract-declarator:
    direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
    direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
    direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**
    direct-abstract-declarator$_{opt}$ **[*]**

function-abstract-declarator:
    direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

Semantics

2 In several contexts, it is necessary to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[198] The optional attribute specifier sequence in a direct abstract declarator appertains to the preceding array or function type. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

3 **EXAMPLE** The constructions

```
(a)     int
(b)     int *
(c)     int *[3]
(d)     int (*)[3]
(e)     int (*)[*]
(f)     int *()
(g)     int (*)(void)
(h)     int (*const [])(unsigned int, ...)
(h)     int (*const [])(unsigned int, ...)
```

name respectively the types (a) **int**, (b) pointer to **int**, (c) array of three pointers to **int**, (d) pointer to an array of three **int** s, ~~(e) pointer to a variable length array of an unspecified number of **int** s,~~ (f) function with no parameter specification returning a pointer to **int**, (g) pointer to function with no parameters returning an **int**, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type **unsigned int** and an unspecified number of other parameters, returning an **int**.

6.7.10 Type definitions

Syntax

1 *typedef-name:*
    identifier

[198] As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification", rather than redundant parentheses around the omitted identifier.

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope. A typedef name shall not be identical to a tag name that has a visible declaration, unless it refers to the type with that same tag.

Semantics

3 In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.8. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A **typedef** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in `T` (known as *T*), and the identifier in `D` has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of `D`. A typedef name shares the same name space as other identifiers declared in ordinary declarators.

4 **NOTE** C and C++ have subtle differences for the rules which identifiers are allowed as type names. The constraint that forbids the reuse of a tag name for another type originates from C++ and we repeat it here, since otherwise programs would not be portable in the common C/C++ core.

5 **EXAMPLE 1** After

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of `distance` is **`int`**, that of `metricp` is "pointer to function with no parameter specification returning **`int`**", and that of `x` and `z` is the specified structure; `zp` is a pointer to such a structure. The object `distance` has a type compatible with any other **`int`** object.

6 **EXAMPLE 2** After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type `t1` and the type pointed to by `tp1` are compatible. Type `t1` is also compatible with type **`struct`** `s1`, but not compatible with the types **`struct`** `s2`, `t2`, the type pointed to by `tp2`, or **`int`**.

7 **EXAMPLE 3** The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
    unsigned t:4;
    const t:5;
    plain r:5;
    unsigned t;
    const t s;
    plain r;
};
```

declare a typedef name `t` with type **`signed int`**, a typedef name `plain` with type **`int`**, and a structure with three ~~bit-field~~ members, one named `t` ~~that contains values in the range [0, 15], an unnamed~~ , an const-qualified ~~bit-field which (if it could be accessed) would contain values in either the range [−15, +15] or [−16, +15]~~member `s` , and one named `r` ~~that contains values in one of the ranges [0, 31], [−15, +15], or [−16, +15]. (The choice of range is implementation-defined.)~~ . The first two ~~bit-field~~ declarations differ in that **`unsigned`** is a type specifier (which forces `t` to be the name of a structure member), while **`const`** is a type qualifier (which modifies `t` which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function `f` is declared with type "function returning **`signed int`** with one unnamed parameter with type pointer to function returning **`signed int`** with one unnamed parameter with type **`signed int`**", and an identifier `t` with type **`long int`**.

8 **EXAMPLE 4** On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **`signal`** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

9 **EXAMPLE 5** If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
      typedef int B[n];   // B is n ints, n evaluated now
      n += 1;
      B a;                // a is n ints, n without += 1
      int b[n];           // a and b are different sizes
      for (int i = 1; i  <  n; i++)
            a[i-1] = b[i];
}
```

6.7.11 Expression types

Syntax

1 *decltype-specifier:*

  `decltype ( ` *identification* **`)`**

  `decltype ( ` *value-expression* **`)`**

identification:

  identifier

  member-access

value-expression:

  expression

Constraints

2 A value expression shall not be an lvalue.[199]

3 The identification or value expression shall be valid and have function or object type. No new type declaration shall be formed by the value expression itself.[200]

[199] As a consequence, **`decltype`** can not be applied to results of array subscript, unary ∗ operator, or of any `()` or **`generic_selection`** primary expression that has an lvalue as result. Note also that the property of being an lvalue or not may differ between C and C++, in particular for prefix increment and decrement operators, assignment operators, the tenary operator and the comma operator.

[200] This could for example happen if the expression contained the forward declaration of a tag type, such as in

Semantics

4 A **decltype** specifier can be used in places where other type specifiers are used to declare or define objects, members or functions. It stands in for the unmodified type of the identification or value expression, even where the expression cannot be used for type inference of its type (opaque types, function types, array types), where a type-qualification should not be dropped, where no other type specifier for the type can be formed (lambda types), or where an identifier may only be accessed for its type without evaluating it (within lambda expressions).

5 If it does not have a variably modified (VM) type, the identification or value expression is not evaluated. For VM types, the same rules for evaluation as for **sizeof** expressions apply. Analogous to **typedef**, a **decltype** specifier does not introduce a new type, it only acts as a placeholder for the type so specified.

6 **NOTE** C++ allows other lvalue expressions as expressions, and the deduced type then is a reference type. Since C does not have references, such a construct should not be used by code that targets the C/C++ core.

7 **EXAMPLE**

```
void f(int);
decltype(f(5)) g(double x) {            // g has type ``void (double)''
  printf("value %g\n", x);
}
decltype(g)* h;                         // h has type ``void (*)(double)''
decltype(true ? g : nullptr) k;         // k has type ``void (*)(double)''

void ell(double A[5], decltype(A)* B); // ell has type ``void (double*, double**)''

extern double D[];                      // D has an incomplete type
decltype(D) C = { 0.7, 99, };           // C has type double[2]
decltype(D) D = { 5, 8.9, 0.1, 99, };  // D is completed to double[4]
decltype(D) E;                          // E has type double[4]
```

For the definition of `g`, the expression `f(5)` has type **void**, and so this becomes the return type. For `h`, the **decltype** specifier uses the identification syntax. Here the function type of `g` stands in for a function type specifier and the result type for `h` is a pointer to function. For `k`, again the expression derivation is used. Here the type is the type of a ternary operator, thus the type of `g` after function to function pointer conversion. As the result, the type of `k` is again a function pointer.

For `ell` the parameter type adjustment takes place before **decltype** specifier is met. Therefore **decltype**`(A)` refers to the type **double**`*` and not to **double**`[5]`.

As for `D`, the type of the expression to which a **decltype** specifier refers has not necessarily to be complete. Here, `C` first inherits that incomplete type but is then completed by the initializer to have type **double**`[2]`. The specification **decltype**`(D)` can then even be used in the definition of `D` itself to complete its type to **double**`[4]`.

6.7.12 Initialization

Syntax

1 *initializer:*

assignment-expression
{ *initializer-list*$_{\text{opt}}$ **}**
{ *initializer-list* **,** **}**

initializer-list:

designation$_{\text{opt}}$ *initializer*
initializer-list **,** *designation*$_{\text{opt}}$ *initializer*

designation:

designator-list **=**

designator-list:

designator

(**struct** newStruct*)0 where **struct** newStruct has not yet been declared, or if it uses a compound literal that declares a new structure or union type in its *type-name* component.

designator-list designator

designator:

[*constant-expression* **]**

. *identifier*

Constraints

2 No initializer shall attempt to provide a value for an object not contained within the entity being initialized. If the type of the object that is initialized is opaque, the initializer shall be omitted or of the form `{}` .

3 The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type.

4 All the expressions in an initializer for an object that has static or thread storage duration shall be constant expressions or string literals.

5 If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

6 If a designator has the form

[*constant-expression* **]**

then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

7 If a designator has the form

. *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

Semantics

8 ~~An~~ For non-opaque object types, an initializer specifies the initial value stored in an object. Unless specified otherwise, for opaque object types an initializer guarantees a valid initial state by setting all bits of the representation to zero.

9 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.

10 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:

— if it has pointer type, it is initialized to a null pointer;

— if it has arithmetic type, it is initialized to (positive or unsigned) zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

— if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

11 The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

12 The rest of this subclause deals with initializers for objects that have aggregate or union type.

13 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union

type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.

14 An array of character type may be initialized by a character string literal ~~or UTF–8 string literal,~~ optionally enclosed in braces. Similarly, an array of **char8_t** may be initialized by an UTF–8 string literal.[201] Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

15 An array with element type compatible with a qualified or unqualified version of **wchar_t**, **char16_t**, or **char32_t** may be initialized by a wide string literal with the corresponding encoding prefix (`L`, `u`, or `U`, respectively), optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

16 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members, which may be empty.

17 Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.[202] In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization then continues forward in order, beginning with the next subobject after that described by the designator.[203]

18 Each designator list begins its description with the current object associated with the closest surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[204] The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

19 The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[205] all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

20 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

21 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.[206]

22 If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.

23 The evaluations of the initialization list expressions are indeterminately sequenced with respect to

[201] For C **char8_t** is a character type and so there may be no difference in the initialization between the two types of string literals. But the distinction is important for the compatibility to C++.

[202] If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

[203] After a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

[204] Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

[205] Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

[206] In particular, if the initializer list is empty all members are initialized implicitly.

one another and thus the order in which any side effects occur is unspecified.[207)]

24 **NOTE** The C language currently does not allow the initializer list to be empty. To ease portability in the C/C++ core, implementations are encouraged to accept such initializers as an extension.

25 **EXAMPLE 1** Provided that `<complex.h>` has been **`#include`**d, the declarations

```
int i = 3.5;
double complex c = 5 + 3 * I;
double complex c = 5 + 3 × I;
```

define and initialize `i` with the value 3 and `c` with the value $5.0 + i3.0$.

26 **EXAMPLE 2** The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

27 **EXAMPLE 3** The declaration

```
int y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of `y` (the array object `y[0]`), namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, so `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`.

28 **EXAMPLE 4** The declaration

```
int z[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `z` as specified and initializes the rest with zeros.

29 **EXAMPLE 5** The declaration

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: `w[0].a[0]` is 1 and `w[1].a[0]` is 2; all the other elements are zero.

30 **EXAMPLE 6** The declaration

```
short q[4][3][2] = {
      { 1 },
      { 2, 3 },
      { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0]` does not begin with a left brace, so up to six items from the current list could be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for `q[1][0]` and `q[2][0]` do not begin with a left brace, so each uses up to six items, initializing their respective

[207)] In particular, the evaluation order need not be the same as the order of subobject initialization.

two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
      1, 0, 0, 0, 0, 0,
      2, 3, 0, 0, 0, 0,
      4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
      {
            { 1 },
      },
      {
            { 2, 3 },
      },
      {
            { 4, 5 },
            { 6 },
      }
};
```

in a fully bracketed form.

31 Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

32 **EXAMPLE 7** One form of initialization that completes array types involves typedef names. Given the declaration

```
typedef int A[];  // OK - declared with block scope
```

the declaration

```
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

33 **EXAMPLE 8** The declaration

```
char s[] = "abc", t[3] = "abc";
```

defines "plain" `char` array objects `s` and `t` whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
     t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines `p` with type "pointer to `char`" and initializes it to point to an object with type "array of `char`" with length 4 whose elements are initialized with a character string literal. If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

34 **EXAMPLE 9** Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
      [member_two] = "member two",
      [member_one] = "member one",
```

```
};
```

35 **EXAMPLE 10** Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = {.quot = 2, .rem = -1 };
```

36 **EXAMPLE 11** Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] =
      { [0].a = {1}, [1].a[0] = 2 };
```

37 **EXAMPLE 12**

```
struct T {
      int k;
      int l;
};

struct S {
      int i;
      struct T t;
};

struct T x = {.l = 43, .k = 42, };

void f(void)
{
      struct S l = { 1, .t = x, .t.l = 41, };
}
```

The value of `l.t.k` is 42, because implicit initialization does not override explicit initialization.

38 **EXAMPLE 13** Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

39 In the above, if `MAX` is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

40 **EXAMPLE 14** Any member of a union can be initialized:

```
union { /* ... */ } u = {.any_member = 42 };
```

~~common definitions (7.19).~~

6.7.13 Type inference

Constraints

1 An underspecified declaration shall contain at least one storage class specifier.

2 For an underspecified declaration of a function that is also a definition, the return type shall be completed as of 6.9.1. For an underspecified declaration of a function that is not a definition and not the declaration of a parameter of a lambda expression, a prior definition of the declared function shall be visible.

3 For an underspecified declaration of an object that is not the declaration of a parameter of a lambda expression, each of its init declarators shall be of one of the forms

declarator **=** *assignment-expression*
declarator **= {** *assignment-expression* **}**
declarator **= {** *assignment-expression* **, }**

such that the declarator does not declare an array.

4 For an underspecified declaration there shall be a type specifier *type* that can be inserted in the declaration immediately after the last storage class specifier that makes the adjusted declaration a valid declaration. All declared objects of that adjusted declaration shall have a type that is the generic type of their assignment expression or a qualified or atomic version thereof. All declared functions shall have a type that is compatible with the type of the corresponding definition.

Description

5 Provided the constraints above are respected, in an underspecified declaration the type of the declared identifiers is the type after the declaration has been adjusted by *type*. The type of each identifier that declares an object is incomplete until the end of the assignment expression that initializes it.

6 **NOTE** The scope of the identifier for which the type is inferred only starts after the end of the initializer (6.2.1), so the assignment expression cannot use the identifier to refer to the object or function that is declared, for example to take its address. Any use of the identifier in the initializer is invalid, even if an entity with the same name exists in an outer scope.

```
{
    double a = 7;
    double b = 9;
    {
        double b = b * b;     // error, RHS uses uninitialized variable
        printf("%g\n", a);    // valid, uses "a" from outer scope, prints 7
        auto a    = a * a;    // error, "a" from outer scope is already shadowed
    }
    {
        auto b    = a * a;    // valid, uses "a" from outer scope
        auto a    = b;        // valid, shadows "a" from outer scope
        ...
        printf("%g\n", a);    // valid, uses "a" from inner scope, prints 49
    }
    ...
}
```

7 **EXAMPLE** Consider the following definitions:

```
static auto a = 3.5;
auto * p = &a;
```

They are interpreted as if they had been written as:

```
static auto double a = 3.5;
auto double * p = &a;
```

which again is equivalent to

```
static double a = 3.5;
double * p = &a;
```

So effectively `a` is a **`double`** and `p` is a **`double`**∗.

```
double A[3] = {};
auto const * pA = A;
auto const (*qA)[3] = &A;
```

Here `pA` is valid because the generic type of `A` is a pointer type, and `qA` is valid because it does not declare an array but a pointer to an array.

6.7.14 Static assertions

Syntax

1 *static_assert-declaration:*

  ~~**_Static_assert**~~ **static_assert** **(** *constant-expression* **,** *string-literal* **)** **;**

  ~~**_Static_assert**~~ **static_assert** **(** *constant-expression* **)** **;**

Constraints

2 The constant expression shall compare unequal to 0.

Semantics

3 The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, if present, except that characters not in the basic source character set are not required to appear in the message.

Forward references: diagnostics (7.2).

6.7.15 Attributes

1 Attributes specify additional information for various source constructs such as types, variables, identifiers, or blocks. They are identified by an *attribute token*, which can either be a *attribute prefixed token* (for implementation-specific attributes) or a *standard attribute* specified by an identifier (for attributes specified in this document).

2 Support for any of the standard attributes specified in this document is implementation-defined and optional. For an attribute token (including an attribute prefixed token) not specified in this document, the behavior is implementation-defined. Any attribute token that is not supported by the implementation is ignored.

3 Attributes are said to appertain to some source construct, identified by the syntactic context where they appear, and for each individual attribute, the corresponding clause constrains the syntactic context in which this appertainance is valid. The attribute specifier sequence appertaining to some source construct shall contain only attributes that are allowed to apply to that source construct.

4 In all aspects of the language, a standard attribute specified by this document as an identifier `attr` and an identifier of the form `__attr__` shall behave the same when used as an attribute token, except for the spelling.[208)]

Recommended practice

5 It is recommended that implementations support all standard attributes as defined in this document.

6.7.15.1 General

Syntax

1 *attribute-specifier-sequence:*

  attribute-specifier-sequence$_{\text{opt}}$ *attribute-specifier*

attribute-specifier:

  ~~**[[** *attribute-list* **]]**~~ ⟦ *attribute-list* ⟧

attribute-list:

  attribute$_{\text{opt}}$

  attribute-list **,** *attribute*$_{\text{opt}}$

attribute:

  attribute-token *attribute-argument-clause*$_{\text{opt}}$

208) Thus, the attributes ~~`[[nodiscard]]`~~ ⟦`nodiscard`⟧ and ~~`[[__nodiscard__]]`~~ ⟦`__nodiscard__`⟧ can be freely interchanged. Implementations are encouraged to behave similarly for attribute tokens (including attribute prefixed tokens) they provide.

attribute-token:
  standard-attribute
  attribute-prefixed-token
standard-attribute:
  identifier

attribute-prefixed-token:
  attribute-prefix ~~**::** *identifier*~~ **::** *attribute-suffix*
attribute-prefix:
  identifier
attribute-suffix:
  identifier
core-attribute:
  core **::** *identifier*

attribute-argument-clause:
  (*balanced-token-sequence*$_{\text{opt}}$ **)**
balanced-token-sequence:
  balanced-token
  balanced-token-sequence balanced-token
balanced-token:
  (*balanced-token-sequence*$_{\text{opt}}$ **)**
  [*balanced-token-sequence*$_{\text{opt}}$ **]**
  { *balanced-token-sequence*$_{\text{opt}}$ **}**
  any token other than a parenthesis, a bracket, or a brace

Constraints

2 The identifier in a standard attribute shall be one of:

`deprecated`	`fallthrough`	`maybe_unused`	`nodiscard`

3 The identifier in a core attribute shall be one of:

`address_independent`	`free`	`realloc`	`state_transparent`
	`idempotent`	`reentrant`	`stateless`
`alias`	`modifies`	`reinterpret`	`unsequenced`
`concurrent`	`noalias`	`state_conserving`	`writethrough`
`evaluates`	`noleak`	`state_invariant`	

Semantics

4 An attribute specifier that contains no attributes has no effect. The order in which attribute tokens appear in an attribute list is not significant. If a keyword (6.4.1) that satisfies the syntactic requirements of an identifier (6.4.2) is contained in an attribute token, it is considered an identifier. A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute.[209)]

5 Additionally, this specification defines attributes for the C/C++ core that have the form of an attribute prefixed token, where the attribute prefix are **core** or **__core__**, and where the identifier is one of the above or the same with prefix and postfix of __. Those forms are equivalent besides their spelling. They are specified in 6.7.15.4 and following.

6 **NOTE** For each standard or core attribute, the form of the balanced token sequence, if any, will be specified.

209) Standard attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.

Recommended Practice

7 Each implementation should choose a distinctive name for the attribute prefix in an attribute prefixed token. Implementations should not define attributes without an attribute prefix unless it is a standard attribute as specified in this document.

8 **EXAMPLE 1** Suppose that an implementation chooses the attribute prefix `hal` and provides specific attributes named `daisy` and `rosie`.

```
[[deprecated, hal::daisy]] double nine1000(double);
[[deprecated]] [[hal::daisy]] double nine1000(double);
[[deprecated]] double nine1000 [[hal::daisy]] (double);
[[deprecated, hal::daisy]] double nine1000(double);
[[deprecated]] [[hal::daisy]] double nine1000(double);
[[deprecated]] double nine1000 [[hal::daisy]] (double);
```

Then all the following declarations should be equivalent aside from the spelling:

```
[[__deprecated__, __hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] [[__hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] double nine1000 [[__hal__::__daisy__]] (double);
[[__deprecated__, __hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] [[__hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] double nine1000 [[__hal__::__daisy__]] (double);
```

These use the alternate spelling that is required for all standard attributes and recommended for prefixed attributes. These may be better-suited for use in header files, where the use of the alternate spelling avoids naming conflicts with user-provided macros.

9 **EXAMPLE 2** For the same implementation, the following two declarations are equivalent, because the ordering inside attribute lists is not important.

```
[[hal::daisy, hal::rosie]] double nine999(double);
[[hal::rosie, hal::daisy]] double nine999(double);
[[hal::daisy, hal::rosie]] double nine999(double);
[[hal::rosie, hal::daisy]] double nine999(double);
```

On the other hand the following two declarations are not equivalent, because the ordering of different attribute specifiers may affect the semantics.

```
[[hal::daisy]] [[hal::rosie]] double nine999(double);
[[hal::rosie]] [[hal::daisy]] double nine999(double); // may have different semantics
[[hal::daisy]] [[hal::rosie]] double nine999(double);
[[hal::rosie]] [[hal::daisy]] double nine999(double); // may have different semantics
```

6.7.15.2 Standard attributes

6.7.15.2.1 The `nodiscard` attribute

Constraint

1 The **`nodiscard`** attribute shall be applied to the identifier in a function ~~declarator~~ declaration or to the definition of a structure, union, or enumeration type. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics

2 A name or entity declared without the **`nodiscard`** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.

Recommended Practice

3 A nodiscard call is a function call expression that calls a function previously declared with attribute **`nodiscard`**, or whose return type is a structure, union, or enumeration type marked with attribute **`nodiscard`**. Evaluation of a nodiscard call as a void expression (6.8.3) is discouraged unless explicitly cast to **`void`**. Implementations are encouraged to issue a diagnostic in such cases. This is typically because immediately discarding the return value of a **`nodiscard`** call has surprising consequences.

4 **EXAMPLE 1**

```
struct [[nodiscard]] error_info { /*...*/ };
struct [[ nodiscard ]] error_info { /*...*/ };
struct error_info enable_missile_safety_mode(void);
void launch_missiles(void);
void test_missiles(void) {
      enable_missile_safety_mode();
      launch_missiles();
}
```

A diagnostic for the call to `enable_missile_safety_mode` is encouraged.

5 **EXAMPLE 2**

```
[[nodiscard]] int important_func(void);
[[ nodiscard ]] int important_func(void);
void call(void) {
      int i = important_func();
}
```

No diagnostic for the call to `important_func` is encouraged despite the value of `i` not being used.

6.7.15.2.2 The `maybe_unused` attribute

Constraint

1 The **`maybe_unused`** attribute shall be applied to the declaration of a structure, a union, a **`typedef`** name, a variable, a structure or union member, a function, an enumeration, or an enumerator. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics

2 The **`maybe_unused`** attribute indicates that a name or entity is possibly intentionally unused. A name or entity declared without the **`maybe_unused`** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

Recommended Practice

3 For an entity marked **`maybe_unused`**, implementations are encouraged not to emit a diagnostic that the entity is unused, or that the entity is used despite the presence of the attribute.

4 **EXAMPLE**

```
[[maybe_unused]] void f([[maybe_unused]] int i) {
      [[maybe_unused]] int j = i + 100;
[[ maybe_unused ]] void f( [[ maybe_unused ]] int i) {
      [[ maybe_unused ]] int j = i + 100;
      assert(j);
}
```

Implementations are encouraged not to diagnose that `j` is unused, whether or not **`NDEBUG`** is defined.

6.7.15.2.3 The `deprecated` attribute

Constraint

1 The **`deprecated`** attribute shall be applied to the declaration of a structure, a union, a **`typedef`** name, a variable, a structure or union member, a function, an enumeration, or an enumerator. It shall appear at most once in each attribute list.

2 If an attribute argument clause is present, it shall have the form:

(*string-literal* **)**

Semantics

3 The **deprecated** attribute can be used to mark names and entities whose use is still allowed, but is discouraged for some reason.[210)]

4 A name or entity declared without the **deprecated** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

Recommended Practice

5 Implementations should use the **deprecated** attribute to produce a diagnostic message in case the program refers to a name or entity other than to declare it, after a declaration that specifies the attribute, when the reference to the name or entity is not within the context of a related deprecated entity. The diagnostic message may include text provided by the string literal within the attribute argument clause of any **deprecated** attribute applied to the name or entity.

6 **EXAMPLE**

```
struct [[deprecated]] S {
struct [[deprecated]] S {
        int a;
};

enum [[deprecated]] E1 {
enum [[deprecated]] E1 {
        one
};

enum E2 {
        two [[deprecated("use 'three' instead")]],
        two [[deprecated("use 'three' instead")]],
        three
};

[[deprecated]] typedef int Foo;
[[deprecated]] typedef int Foo;

void f1(struct S s) { // Diagnose use of S
        int i = one; // Diagnose use of E1
        int j = two; // Diagnose use of two: "use 'three' instead"
        int k = three;
        Foo f; // Diagnose use of Foo
}

[[deprecated]] void f2(struct S s) {
[[deprecated]] void f2(struct S s) {
        int i = one;
        int j = two;
        int k = three;
        Foo f;
}

struct [[deprecated]] T {
struct [[deprecated]] T {
        Foo f;
        struct S s;
};
```

Implementations are encouraged to diagnose the use of deprecated entities within a context which is not itself deprecated, as indicated for function `f1`, but not to diagnose within function `f2` and `struct T`, as they are themselves deprecated.

[210)] In particular, **deprecated** is appropriate for names and entities that are obsolescent, insecure, unsafe, or otherwise unfit for purpose.

6.7.15.2.4 The `fallthrough` attribute

Constraint

1 The attribute token **`fallthrough`** shall only appear in an attribute declaration (6.7); such a declaration is a *fallthrough declaration*. The attribute token **`fallthrough`** shall appear at most once in each attribute list and no attribute argument clause shall be present. A fallthrough declaration may only appear within an enclosing **`switch`** statement (6.8.4.2). The next ~~statement that would be executed~~ block item (6.8.2) after a fallthrough declaration shall be a labeled statement whose label is a **`case`** label or **`default`** label for the same **`switch`** statement.

Recommended Practice

2 The use of a fallthrough declaration is intended to suppress a diagnostic that an implementation might otherwise issue for a **`case`** or **`default`** label that is reachable from another **`case`** or **`default`** label along some path of execution. Implementations are encouraged to issue a diagnostic if a fallthrough declaration is not dynamically reachable.

3 **EXAMPLE**

```
void f(int n) {
      void g(void), h(void), i(void);
      switch (n) {
      case 1: /* diagnostic on fallthrough discouraged */
      case 2:
            g();
            [[fallthrough]];
            [[ fallthrough ]] ;
      case 3: /* diagnostic on fallthrough discouraged */
            h();
      case 4: /* fallthrough diagnostic encouraged */
            i();
            [[fallthrough]]; /* constraint violation */
            [[ fallthrough ]] ; /* constraint violation */
      }
}
```

6.7.15.3 Core storage attributes

Syntax

1 *core-storage-attribute:*
  `core` :: *identifier attribute-argument-clause*$_{opt}$

Constraints

2 The identifier in a core storage attribute shall be one of

`alias`	**`noalias`**	**`writethrough`**
`free`	**`realloc`**	

3 Unless specified otherwise, the core storage attributes shall only be applied to an object or function declaration, to a member declaration, to an identifier in a direct declarator, to a function declarator, to a lambda expression, to a pointer declarator, or, in a type specifier qualifier list.

4 If they are applied in a type specifier qualifier list, they shall follow a **`typedef`** name or a **`decltype`** specifier that stands in for a pointer type. If they are applied to a function declarator or a lambda expression, the return type, which is possibly inferred, shall be a pointer type.

5 If they are applied to a object or function declaration, the effect shall be as if they are applied to the corresponding identifier. If they are applied to an identifier, that identifier shall be a function or object. If they are applied to a union or structure member or to an identifier that has an object type, the type shall not be opaque or atomic nor an array with such a base type.

Description

6 The intended use of the core storage attributes is to promote optimization, and deleting all instances of the attributes from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior),[211)] with the notable exception for the case that the `core::noalias` attribute is used for an identifier with external linkage.

In the following a function or lambda is said to be an *allocator* function if has has a pointer return value that has an implicit or explicit `core::noalias` attribute; it is said to be a *deallocator* if it has a `core::free` or a `core::realloc` parameter. A function or lambda is said to *allocate* a storage instance if it calls an allocator, and said to *deallocate* a storage instance if it calls a deallocator.

6.7.15.3.1 The `core::noalias` attribute

Constraints

1 Additional constraints to the above apply. The attribute argument list shall be omitted or of the form

(*expression* **)**

where the expression has integer type. For the evaluation of the expression, the same rules as for the evaluation of array sizes apply.

2 If the `core::noalias` attribute is applied to an identifier additional constraints apply. If it is applied to an identifier in a declaration, it shall be applied to all declarations (including a definition) in the same translation unit.

3 If it is a function, the unary & operator shall not be applied and an implicit function to pointer conversion shall only be formed if it is used as the left operand of a function call operator.

4 If it is an object, the unary & operator shall not be applied to the object or any of its elements or members, even recursively, and an implicit array to pointer conversion shall only be formed if it is implied by an array subscript operator for which the size expression is an integer constant expression.

5 If the `core::noalias` attribute is applied to the declaration of a union or structure member *name* of union or structure *S*, the rules for objects apply to all member access designations that use an lvalue `s` of type *S* (`s.`*name*) or a pointer `ps` to *S* (`ps`→*name*), and then recursively to all their elements or members.

Description

6 If an attribute argument list is provided, the expression (called the *size* of the attribute) shall be strictly positive. In cases where the attribute is applied to an identifier or to the declaration of a union or structure member, the size shall be omitted.

7 In the case the `core::noalias` attribute is applied to the declarator of a pointer type `T*`, a size n indicates that the pointer will be used to access an array of type `T[`n`]`. If it is omitted the attribute is said to have unknown size and the pointer gives access to an incomplete array of type `T[]`.

8 If the `core::noalias` attribute is applied to an identifier or declaration, it specifies that the address of the object or function will never be taken. Additionally, for any definition of an identifier to which the `core::noalias` attribute is applied the following properties hold:

- If it is an object definition, that object will never alias with any other object.
- If it is an object with automatic storage duration, it will never escape its defining scope.
- If it is an object or function with internal linkage, it will never escape the translation unit in which it is defined.

If it is an inline constant or function, no external definition shall be required.

[211)] Two translation units where one has been translated by ignoring the attribute and the other by taking it into account can be linked into one executable. In particular, applying these attributes to declarations of structure members may change the layout.

9 If the `core::noalias` attribute is applied to a declaration of a member of a union or structure, it specifies that for any object with that union or structure type, the address of the member will never be taken. The alignment restrictions for such a member may be looser than the alignment restrictions for other objects of the same type as the member, but they shall be the same for all such members of the same type that have the `core::noalias` attribute . Such a member will never alias with any other object of the same type as the member, unless both are members of objects of the same union or structure type, and these containing objects alias. The start address of the member, however obtained, shall not be converted to a pointer to the type of the member.[212)]

10 If a `core::noalias` attribute is applied to an identifier or declaration of a function parameter that is specified in array notation with an array size expression E, the attribute is applied twice, to the identifier of the parameter, if any, with no attribute argument list and to the pointer type that results from the array parameter rewrite, propagating the size expression E to the attribute. In that case E shall evaluate to a value that is greater than 0.

11 If it is applied to a function declarator or a lambda expression, the effect is as if the pointer return type has the `core::noalias` attribute with the same size.[213)] If it is applied to the pointer return type of a function or lambda it indicates that a non-null pointer value that is returned by any call to that function refers to the first element of an array object (of type `T[`n`]` or `T[]`) as above, that has not been encountered before and that will thus not alias with any known object[214)]

12 If the `core::noalias` attribute is applied to a declarator of pointer type, it reflects a specific property of a pointer value that is accessible through the declaration. An object that is accessed through a noalias pointer has a special association with that pointer. This association, defined below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.[215)]

13 Let `D` be a declaration of an ordinary identifier that provides a means of designating an object `P` as a pointer to type `T` having the `core::noalias` attribute with unknown size.

14 If `D` appears inside a block and does not have storage class **`extern`**, let `B` denote the block. If `D` appears in the list of parameter declarations of a function definition, let `B` denote the associated block. Otherwise, let `B` denote the block of **`main`** (or the block of whatever function is called at program startup in a freestanding environment).

15 In what follows, a pointer expression `E` is said to be *based* on object `P` if (at some sequence point in the execution of `B` prior to the evaluation of `E`) modifying `P` to point to a copy of the array object into which it formerly pointed would change the value of `E`.[216)] Note that "based" is defined only for expressions with pointer types.

16 During each execution of `B`, let `L` be any lvalue that has `&L` based on `P`. If `L` is used to access the value of the object `X` that it designates, and `X` is also modified (by any means), then the following requirements apply: `T` shall not be const-qualified. Every other lvalue used to access the value of `X` shall also have its address based on `P`. Every access that modifies `X` shall be considered also to modify `P`, for the purposes of this subclause. If `P` is assigned the value of a pointer expression `E` that is based on another noalias pointer object `P2`, associated with block `B2`, then either the execution of `B2` shall begin before the execution of `B`, or the execution of `B2` shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

17 Here an execution of `B` means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with `B`.

18 If `P` is as above, but the attribute provides a size n, additional restrictions apply. Any expression `E` that is based on `P` shall only access bytes in the array of type `T[`n`]` that is associated to `P`. If `Q` is

[212)] These relaxed alignment properties allow implementations to pack such members with less padding.

[213)] This allows to effectively associate a `core::noalias` attribute to a pointer return value by using a size expression that uses the names of parameters.

[214)] This typically indicates that the function behaves similar to the library function **`malloc`**.

[215)] For example, a statement that assigns a value returned by **`malloc`** to a single pointer establishes this association between the allocated object and the pointer.

[216)] In other words, `E` depends on the value of `P` itself rather than on the value of an object referenced indirectly through `P`. For example, if identifier `p` has type `(int **[[core::noalias]])`, then the pointer expressions `p` and `p+1` are based on the noalias pointer object designated by `p`, but the pointer expressions `*p` and `p[1]` are not.

another pointer of type S with a **core**::**noalias** attribute of size m, and that is associated with the same block B, then the two arrays to which P and an Q refer (of type T[n] or S[m], respectively) shall share no representation byte.

Recommended Practice

19 It is recommended that applications that use the **core**::**noalias** attribute for a pointer return value also assert the **core**::**noleak** attribute, see below.

20 **EXAMPLE 1** Suppose that **double** has an alignment requirement of 8 and consider the following structures:

```
    struct S {
      char indicator;
      double field;
    };
    struct T {
      char indicateur;
      [[ core::noalias ]] double champs;
    };
```

Then S would necessarily have **offsetof**(S, field) as 8 or more, and **sizeof**(S) as 16 or more. For T the implementation could chose differently, for example an alignment of 4. Then, **offsetof**(T, champs) can be 4, and **sizeof**(T) can be 12.

21 Such an alignment then cannot lead to pointer misalignment, because the unary & cannot be applied to the member champs. On the other hand, there may be a tradeoff for the gain in size because load or store operations to the champs member may be more expensive.

22 **EXAMPLE 2** The following shows a declaration with parameters in array notation and its equivalent rewrite.

```
    void add( [[ core::noalias ]] double a[3][4],
              [[ core::noalias ]] double (*b)[4]);

    void add(double (* [[ core::noalias(3) ]] a [[ core::noalias ]] )[4],
             double (* b [[ core::noalias ]] )[4]);
```

23 **EXAMPLE 3** The example function date_alloc from above returns a pointer to a string that has been freshly allocated. So a **core**::**noalias** attribute can be added indicating that the returned array will not alias and also to provide information about the array size.

```
    #include <time.h>
    #include <stdlib.h>
    char const* date_alloc(void) [[ core::noalias(26) ]] {
        char* [[ core::noalias(26) ]] ret = malloc(26);
        if (ret) ctime_r(time(nullptr), ret);
        return ret;
    }
```

24 **EXAMPLE 4** The file scope declarations

```
int* [[ core::noalias ]] a;
int* [[ core::noalias ]] b;
extern int c[];
```

assert that if an object is accessed using one of a, b, or c, and that object is modified anywhere in the program, then it is never accessed using either of the other two. Because no sizes are indicated, the extent of the access through the pointers cannot be verified by the translator, and the programmer has to ensure the necessary assertions by other means.

25 **EXAMPLE 5** The function parameter declarations in the following example

```
void f(int n, int* [[ core::noalias ]] p, int* [[ core::noalias ]] q)
{
    while (n-- > 0)
        *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other. The translator can make this no-aliasing inference based on the parameter declarations alone, without analyzing the function body.

26 It cannot, though, assert that the function body conforms to these guarantees. This can be achieved by providing more information directly or indirectly to the attributes. There are two possibilities, to provide the size information. The first is to simply add the size to the attribute:

```
void f0(int n, int* [[core::noalias(n)]] p, int* [[core::noalias(n)]] q);
```

Even better would be to use the array size where it is "natural" and have the whole function definition as

```
void f1(int n, [[core::noalias]] int p[n], [[core::noalias]] int q[n])
{
    while (n-- > 0)
        *p++ = *q++;
}
```

This is a short form of overall four attributes when the parameter declarations are rewritten to

```
void f1(int n, int* [[core::noalias(n)]] p [[core::noalias]],
    int* [[core::noalias(n)]] q [[core::noalias]]);
```

27 The benefit of the **core::noalias** attributes is that they enable a translator to make an effective dependence analysis of function `f` without examining any of the calls of `f` in the program. The cost is that the programmer has to examine all of those calls to ensure that they have defined behavior. For example, the second call of `f` in `g` has undefined behavior because each of `d[1]` through `d[49]` is accessed through both `p` and `q`.

```
void g(void)
{
    extern int d[100];
    f(50, d + 50, d); // valid
    f(50, d + 1, d); // undefined behavior
}
```

28 Providing sizes to the attributes improves on that situation, by making many of such invalid calls diagnosable. For example, the second call of `f0` in `g0` has the same undefined behavior as the second call of `f` above, but the translator may perform data flow analysis and detect that arrays provided by the second and third argument overlap.

```
void g0(void)
{
    extern int d[100];
    f0(50, d + 50, d); // valid
    f0(50, d + 1, d); // undefined behavior, diagnosed
}
```

29 **EXAMPLE 6** The function parameter declarations

```
void h(int n, int* [[core::noalias]] p, int* [[core::noalias]] q, int* [[core::noalias]] r)
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two noalias pointers. In particular, if `a` and `b` are disjoint arrays, a call of the form `h(100, a, b, b)` has defined behavior, because array `b` is not modified within function `h`.

30 Nevertheless, a declaration (and definition)

```
void h(int n, [[core::noalias]] int p[n], int const q[const n], int const r[const n])
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i] + r[i];
```

```
}
```

would be better suited to reflect the requirements to the function and also for its callers. For the calling side and the translation of the function body itself, it can be concluded that `p` cannot alias neither `q` nor `r`, and that `q` and `r` may refer to arrays that totally or partially overlap with each other.

31 **EXAMPLE 7** The rule limiting assignments between noalias pointers does not distinguish between a function call and an equivalent nested block. With one exception, only "outer-to-inner" assignments between noalias pointers declared in nested blocks have defined behavior.

```
{
    int* [[core::noalias]] p1;
    int* [[core::noalias]] q1;
    p1 = q1; // undefined behavior
    {
        int* [[core::noalias]] p2 = p1; // valid
        int* [[core::noalias]] q2 = q1; // valid
        p1 = q2;                        // undefined behavior
        p2 = q2;                        // undefined behavior
    }
}
```

32 The one exception allows the value of a noalias pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`.

```
typedef struct { int n; float* [[core::noalias]] v; } vector;
vector new_vector(int n)
{
    vector t = {
        .n = n,
        .v = malloc(sizeof(float[n])),
    };
    return t;
}
```

33 **EXAMPLE 8** Suppose that a programmer knows that references of the form `p[i]` and `q[j]` are never aliases in the body of a function:

```
void f(int n, int *p, int *q) { /* ... */ }
```

but that not more information about the array sizes that are accessed is available. There are several ways that this information could be conveyed to a translator using the **core::noalias** attribute without using a size. Example 6.7.15.3.1 ex. 5 shows the most effective way in that situation, attributing all pointer parameters, and can be used provided that neither `p` nor `q` becomes based on the other in the function body. A potentially effective alternative is:

```
void f(int n, int* [[core::noalias]] p, int * const q) { /* ... */ }
```

Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use subtler reasoning: that the const-qualification of `q` precludes it becoming based on `p`. There is also a requirement that `q` is not modified, so this alternative cannot be used for the function in Example 6.7.15.3.1 ex. 5, as written.

34 **EXAMPLE 9** Another potentially effective alternative is:

```
void f(int n, int *p, int const* [[core::noalias]] q) { /* ... */ }
```

Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use even subtler reasoning: that this combination of the **core::noalias** attribute and **const** means that objects referenced using `q` cannot be modified, and so no modified object can be referenced using both `p` and `q`.

35 **EXAMPLE 10** The least effective alternative is:

```
void f(int n, int* [[core::noalias]] p, int *q) { /* ... */ }
```

Here the translator can make the no-aliasing inference only by analyzing the body of the function and proving that `q` cannot become based on `p`. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator is required to assume that aliases are present because assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and thus need to conservatively assume that aliases are present.

6.7.15.3.2 The `core::alias` attribute

Constraints

1 Additional constraints to the above apply. The attribute argument list shall be omitted or of the form

(*identifier* **)**

where the identifier is called the *aliased symbol* . It shall only be omitted, if it is applied to the pointer return value of a function or lambda, or to a member declaration.

2 If the `core::alias` attribute is applied to the name of an object or function, it shall have external linkage, the aliased symbol shall be visible at the declaration, and shall have internal linkage. If it is an object, the aliased symbol shall also have object type, be representable by the type of the identifier, and shall not be the aliased symbol of a different identifier. If it is a function, the aliased symbol shall also have function type and the two types shall be compatible. The translation unit shall not provide an explicit external definition for the identifier, and neither shall any other translation unit, if the feature macro `__CORE_ALIAS_OVERWRITES__` (6.10.8.1) evaluates to `true`.

3 If the `core::alias` attribute is applied to a the member declaration of a union or structure, the `core::noalias` attribute with all its constraints is implied.

4 If the `core::alias` attribute is applied to a declarator,[217] the identifier shall have pointer to object type, and the generic type of the aliased symbol, if any, shall be a pointer type.

Description

5 The `core::alias` attribute serves to identify exceptions of the aliasing rules, in particular to establish that a pointer value is based on another pointer, where the translator may not be able to deduce such a based-on relation automatically, or to indicate that a member of a structure or union potentially shares a storage unit with other members.

6 If the `core::alias` attribute is applied to an identifier with linkage, the declaration stands in for an external definition of the identifier. If an explicit external definition of the same identifier in another translation unit is permitted (`__CORE_ALIAS_OVERWRITES__` is `false`), such an explicit definition is the external definition that is visible to all other translation units.

7 If the identifier is a function, the function and the aliased symbol stand in for each other and implicit and explicit conversion to a function pointer of the identifier or of the aliased symbol results in the same address. If the identifier is an object, the object and the aliased symbol share the same storage instance, and any change of value or state of one affects the other.

8 If the `core::alias` attribute is applied to the declaration of members of a union or structure, the constraints on the layout of the union or structure are even further relaxed than implied by the implicit `core::noalias` attribute. A *union pack* or *structure pack* is a maximal set of consecutive members $m_0, \ldots, m_k$ of the same union or structure declaration, respectively, that all have been declared with the `core::alias` attribute . The effect is as if the whole *pack* is represented by a byte array α that uses an implementation-defined internal representation for the members $m_0, \ldots, m_k$, and that the attributes of all the declarations of these members were rewritten to `core::alias(`α`)`, if such an attribute could be formed.[218] For a union pack, the size of α and representation of the members in it shall only depend on the representations of the types of the members; for a structure pack the representation additional depends on the declaration order of the members. [219]

[217] By the above the declarator has pointer type.

[218] Changing any byte in α may change the value of any of the members.

[219] The size of α can be much smaller that the sum of the sizes of the underlying types. Nevertheless, packs of different structure types that contain a sequence of equivalently represented members in the same order have equivalent representations.

9 If the **core**::**alias** attribute is applied to a declarator and the generic type of the aliased symbol is pointer to object type, the so annotated pointer value is based-on the aliased symbol. If the generic type of the aliased symbol has function pointer type, the so annotated pointer value is based on a not further specified internal state of the aliased function.

10 If the **core**::**alias** attribute is applied to the pointer return value of a function and the attribute argument list is omitted, the effect is as if it had been given the name of the function as aliased symbol. If the **core**::**alias** attribute is applied to the pointer return value of a lambda expression and the attribute argument list is omitted, the meaning is similar, but the returned pointer is based on an internal state of the lambda expression.[220)]

11 **EXAMPLE 1** A structure that contains several Boolean flags and some color values could look as follows:

```
struct U {
  size_t size;
   [[ core::alias ]] bool even, sign;
   [[ core::alias ]] unsigned char red, green, blue;
   [[ core::alias ]] bool done, simple;
};
```

The six members form a structure *pack* and are be grouped together in one anonymous byte array for their presentation. This array only represents `4 + 3×CHAR_BIT` bits of information, so generally it only needs 4 bytes, but the details are left to the implementation.

12 An equivalent definition of `struct U` with bit-field notation would be:

```
struct U {
  size_t size;
  bool even:1, sign:1;
  unsigned red:CHAR_BIT, green:CHAR_BIT, blue:CHAR_BIT;
  bool done:1, simple:1;
};
```

13 To ensure that the different groups are placed into separate storage units, they may be placed into separate anonymous structures. Because packs end with the first structure declaration that contains them, this forces the formation of three different structure packs:

```
struct V {
  size_t size;
  struct {
     [[ core::alias ]] bool even, sign;
  } ; /* anonymous structure β0 */
  struct {
     [[ core::alias ]] unsigned char red, green, blue;
  } ; /* anonymous structure β1 */
  struct {
     [[ core::alias ]] bool done, simple;
  } ; /* anonymous structure β2 */
};
```

Here, β_0 and β_2 would need at least one byte each and β_1 three, and so the three anonymous structures would need at least 5 bytes.

14 An equivalent definition of `struct V` with bit-field notation would be:

```
struct V {
  size_t size;
  bool even:1, sign:1;
  bool :0;                                              // separates packs
```

220) So the state is identified with the lambda expression, and not the lambda value. All possible copies of the lambda value refer to the same internal state.

```
    unsigned red:CHAR_BIT, green:CHAR_BIT, blue:CHAR_BIT;
    bool :0;                                            // separates packs
    bool done:1, simple:1;
  };
```

15 **EXAMPLE 2** To illustrate the impact of these attributes on pointers that are returned from functions, consider the declarations of the following library functions.

```
char * [[core::alias]] asctime( [[core::noalias]] const struct tm timeptr[1]);

char *asctime_r( [[core::noalias]] const struct tm timeptr[1],
                 [[core::noalias]] char buf[26]) [[core::alias(buf)]];

char * [[core::alias]] tmpnam(char *s) [[core::alias(s), core::modifies(fopen)]];

void * realloc(void *ptr, size_t size) [[core::noalias(size), core::modifies(malloc)]];
```

Here, for **asctime_r**, **tmpnam** and **realloc** the possibility of placing the attributes after the parameter list is important, because when the return type is specified, information that is needed for the attribute is not yet available, namely the parameters buf, s and size, respectively.

16 For the aliasing properties, the attributes instruct the translator that **asctime** returns a value that is based on some hidden static state in the function. Thus if there are several visible calls to that same function (or to **ctime** that uses the same state) the translator knows that their return values may alias.

17 For **asctime_r**, the translator knows that the return value aliasses with one of the arguments, and that it thus has to be careful when modifying this buffer. Nevertheless, it may not infer size information from the **core::alias** attribute . The only information that is provided is that the return value aliasses with buf in some way, not that the pointer value is necessarily the same. This is for example the case for the **strtok** library function that may return a pointer to some byte in the argument s1.

18 For **tmpnam**, the situation is even more complicated, as there are two **core::alias** attributes. This is so, because the function may either return the argument, or, a pointer to a **static** buffer. Regardless which occurs, the translator is warned that aliasing with some other object might happen.

19 For **realloc** the effect is the opposite. The translator knows that the return value has to be considered to refer to a new storage instance, that has not been met before, and it disposes even of the size of that storage instance. In particular, it knows, if the return is not a null pointer, that ptr is the address of another storage unit than the argument.

6.7.15.3.3 The core::free attribute

Constraints

1 The **core::free** attribute shall be applied to a function parameter of pointer type or to a pointer type declarator of such a function parameter.

Description

2 The **core::free** attribute indicates that a function or lambda is a deallocator, that is, with respect to the parameter, the function behaves like the **free** library function. Such a parameter p shall previously have been allocated, that is be a **core::noalias** return value, and the associated provenance *I* to which p refers shall not be accessed after that call. All pointer values referring to *I* are henceforth indeterminate, and any other valid pointer q refers to a storage instance *J* that is unaffected by modifications that have been applied to *I*.

Recommended Practice

3 It is recommended that implementations diagnose any of the following situations if they may:

— Use of a pointer argument to such a parameter that has not been obtained as a **core::noalias** return.

— Use of a pointer to the same storage instance *I* after the return from the function, in particular the use of the same argument for a second call to a deallocator.

4 It is recommended that applications that use the **core::free** attribute for a pointer parameter also assert the **core::noleak** attribute, see below.

6.7.15.3.4 The `core::realloc` attribute

Constraints

1 The `core::realloc` attribute shall be applied to a function parameter of pointer type or to a pointer type declarator of such a function parameter. It shall be applied to at most one parameter of the same function.

2 A function with such a parameter shall return a pointer type, and an implicit `core::noalias` attribute is assumed for that return value.

Description

3 The `core::realloc` attribute indicates that the function or lambda is simultaneously a deallocator and an allocator and, in particular, that with respect to the parameter and the return value, the function behaves like the `realloc` library function. Such a parameter `p` shall previously have been allocated as a `core::noalias` return value. The reallocation operation is considered to be successful if the function returns a non-null pointer, in which case the same requirements as for the `core::free` attribute apply; it is considered to have failed if the function returns null, and in that case the pointer value `p` and the storage instance *I* remain available after the call.

Recommended Practice

4 It is recommended that implementations diagnose any of the following situations if they may:

— Use of a pointer argument to such a parameter that has not been obtained as a `core::noalias` return.

— Use of a pointer to the same storage instance *I* after the return from the function, in particular the use of the same argument for a second call to a deallocator, unless the call is known to have failed.

5 It is recommended that applications that use the `core::realloc` attribute for a pointer parameter also assert the `core::noleak` attribute, see below.

6 **NOTE** The conventions for the `core::realloc` attribute are not suited for functions that deallocate the storage instance without allocating a new one. The `core::free` attribute is better suited for such situations.

Forward references: storage management functions (7.22.4).

6.7.15.3.5 The `core::writethrough` attribute

Constraints

1 The `core::writethrough` attribute shall be applied to a declaration of an object or function, to a pointer declarator, to a function declarator, or to a lambda expression. If it is applied to the declaration of a function, to a function declarator or to a lambda expression, the return type shall be a pointer type.

Description

2 A *writethrough* operation is a store operation on an lvalue that does not convert the lvalue and that discards any previous value of the underlying object, if any. Initialization, simple assignment (6.5.17.1),[221] bytewise copy, and several library operations[222] perform such writethrough operations. If the `core::writethrough` attribute is applied to the declaration of an object, it defines a requirement for the corresponding initial lvalue; similarly, if it is applied to a pointer declarator of a pointer object it defines a requirement for the lvalue to which the initial pointer value refers, whenever the definition of the pointer object is met.[223] The requirement is that such an lvalue or any of its subobjects or representation bytes shall not be converted to a value before a value has been stored by one or several writethrough operations into the object, subobject or representation

[221] Initialization and simple assignment are the only writethrough operations that can be performed on addressless objects.

[222] The library operations that constitute writethrough operations are for example most copying functions (7.24.2) including `va_copy`, specific initialization functions (`atomic_init`, `cnd_init`, `memset`, `mtx_init`, `thrd_create`, `tss_create`), some "get" or formatting functions (`fegetexceptflag`, `snprintf`, `totext`, `timespec_get`), and uncoditional atomic store operations (`atomic_store`, `atomic_store_explicit`, `atomic_flag_clear`, `atomic_flag_clear_explicit`).

[223] Thus, if the pointer object is a function parameter, the requirement applies each time the function is called.

byte that is evaluated. If the **core::writethrough** attribute is applied to the declaration of an array parameter, the effect is the same as if the adjusted pointer declarator had the **core::writethrough** attribute .

3 Additionally, if the attribute is applied to a function parameter of array type or to the pointer declarator of a function parameter, every call to the function or lambda shall perform one or several store operations, such that the whole lvalue that is designated by the array or pointer parameter is completely rewritten after the call.[224)]

4 If the **core::writethrough** attribute is applied to the pointer declarator of a function return type, to a function declaration, to a function declarator or to a lambda expression, it indicates that the so-returned pointer refers to an lvalue that is indeterminate. Such an lvalue shall not be converted before it is otherwise initialized.

Recommended Practice

5 Whenever a data flow dependency permits, it is recommended that implementations diagnose a possible conversion of the refered-to lvalue of a pointer parameter or return value with the **core::writethrough** attribute that violates the above requirements.

6 In contrast to that, if the **core::writethrough** attribute is applied to an object the implementation may assume that each lvalue conversion of that object is valid and should not diagnose circumstantial control flow that might violate the requirements.

7 **EXAMPLE** The following declarations show different usages of the **core::writethrough** attribute .

```
[[core::writethrough]] double ε;                  // we promis to initialize ε
double f([[core::writethrough]] double x, int i); // the value of x will be ignored by f
double g(double* [[core::writethrough]] y);        // (*y) will be initalized
double h([[core::writethrough]] double z[45]);    // 45 elements will be initalized

// size indeterminate elements will be allocated
double* k(size_t size) [[core::writethrough, core::noalias(sizeof(double[size]))]];

// size elements will be initialized and returned
double* l(size_t size, [[core::writethrough]] double A[size]) [[core::alias(A)]];

double q(double* [[core::writethrough]] y){
  switch (theMoonShines) {
    case steady: *y = π;
  }
  return *y;                 // diagnose: might be uninitialized
                             // diagnose: *y should be initialized
}

_Noreturn r(double* [[core::writethrough]] y) {
  exit(10);                  // diagnose: *y should be initialized
}

_Noreturn s(double* [[core::writethrough]] y) {
  abort();                  // do not diagnose
}
```

Here, in q the return expression violates the requirements twofold: first it might use an indeterminate value for an lvalue conversion, second the return statement represents a possible control flow that does not initialize *y.

8 The difference between r and s is the possible future use of *y. For the first there may still be application code, namely **exit** handlers, that are executed after the call to **exit**. So the translator cannot know if *y will be refered to. For s this consideration does not apply because **abort** will terminate the execution without calling any application code.

6.7.15.4 Core function attributes

Syntax

224) Here, the array bounds, if any, before the adjustment to a pointer type indicate the number of elements that are assumed to be initialized by the call.

1 *core-function-attribute:*
  core :: *identifier attribute-argument-clause*$_{opt}$

Constraints

2 The identifier in a core function attribute shall be one of

`address_independent`	`evaluates`	`noleak`	`state_invariant`
`alias`	`idempotent`	`reentrant`	`state_transparent`
`concurrent`	`modifies`	`reinterpret`	`stateless`
	`noalias`	`state_conserving`	`unsequenced`

3 Unless specified otherwise, the attribute tokens of core function attributes shall only appear in the attribute specifier sequence that appartains to a function declarator or a lambda expression.[225] If they appear in a function declarator, the function declarator shall be used to declare a function, and the corresponding attribute is a property of the function itself and not of its type. If they appear in a lambda expression, the attribute becomes a property of the resulting lambda value and is propagated to any copy of the lambda value. The attribute tokens shall appear at most once in each attribute list. Unless stated otherwise, the attribute argument clause shall be omitted.

4 For a given attribute token and a declaration of a function with that attribute the following constraints hold.

— If the translation unit forms the definition of the function, that definition shall have the attribute and the attribute argument clause, if any, shall be identical.

— Otherwise, if the function has external linkage and refers to a definition in another translation unit that does not have the attribute or with a different attribute argument clause, the behavior is undefined.

5 Unless stated otherwise, if the attribute is applied to the definition of a function or to a lambda expression, any lambda expression, lambda value, or function specifier that is evaluated within the function body, shall have the same attribute.

Description

6 The attributes described are not part of the prototype of a such annotated function, and the knowledge about the attribute might get lost when forming a function pointer, and, in particular, when passing such a function pointer between different translation units. Lambda values provide more insurance that the attribute information is always attached; they have no declaration syntax, and so a complete chain of type inference will always lead back to the evaluation of the lambda expression. Thus, the sought properties by the annotation with such attributes are only effectively diagnosable if the designator in a function call is either the name of a function or a lambda value.

7 Unless stated otherwise, if a function attribute is applied to the definition of a function or to a lambda expression, any lambda or function specifier that is called in the function body via a function pointer shall have the same attribute.

8 The attributes defined in this clause provide optimization opportunities for functions and lambdas. Their main goal is to provide the translator with information about the access of functions and objects coming from surrounding scopes and such that it may deduce certified properties. This certification is ensured by forcing the attributes to be consistently present for a function definition if any declaration has them, and to force the same type of attributes on other functions or lambdas that are called in the function body.

9 A first pair of attributes, **core**::**noleak** and **core**::**address_independent**, makes assertions about the behavior of functions with respect to the address space. The first guarantees that the function will not leak any allocation, that is, that every newly allocated storage instance will either be

[225] That is, they appear right after the parameter list, if any, and before the function body or semicolon.

deallocated within the same function call, or a pointer to it will be returned as a **core::noalias** pointer. The second, forbids any exposure of storage instances or synthesis of pointers, and thus guarantees that the execution of the function is independent of any properties of the address space or of any particular address choices of any specific execution.

10 One set of attributes, **core::evaluates** and **core::modifies**, works with visible identifiers and establishes a strict framework of data flow from static or thread-local objects in and out of the function body. In addition, the **core::stateless** attribute guarantees that a function or lambda can not hold hidden state in form of a local static or thread-local variable. The second set, **core::state_invariant**, **core::state_conserving** and **core::state_transparent** go beyond this by controlling not only which identifiers are accessed directly, but also which objects are accessed through pointer indirections. Then, there are **core::idempotent**, **core::independent** and **core::unsequenced**, that are the most interesting attributes for optimization, but which can themselves not easily asserted through syntax and strong typing.

11 The attributes **core::evaluates** and **core::modifies** allow to specify exactly the identifiers that may be used by a function or lambda to receive and provide information. These *channels* of the function or lambda are the parameters, the captures (lambdas only), and the accessed identifiers with internal or external linkage, that is identifiers that have static or thread-local storage duration that are accessed. Additionally, some execution state that is defined in library clauses are specially named. Together with a possibly empty set of implementation-defined identifiers, they form the *C library channels*. C library channels shall only be used in one of these attributes if the corresponding headers have been included:

— **errno** as of `<errno.h>` (7.5) is interpreted as if it were implemented as a thread-local variable of type **int**.

— **math_errhandling** as of `<math.h>` (7.12) is interpreted as if it were implemented as an object with extern linkage and with type **const int**.

— **stdin**, **stdout**, or **stderr** as of `<stdio.h>` (7.21) are interpreted as if they were implemented as objects with extern linkage of type **FILE**∗.

— **fopen** is a placeholder for the global file acces state used by **fopen** and similar functions `<stdio.h>` (7.21) or other implementation-specific functions that handle files.

— **fenv** is a placeholder for the thread-local floating-point environment as used by the functions in headers `<fenv.h>` (7.6), `<math.h>` (7.12), and `<complex.h>` (7.3).

— **time** is a placeholder for the overall shared time environment for the the time manipulation functions as of `<time.h>` (7.27.2) or other implementation-specific functions that access time. The **time** channel behaves as if it is a **volatile** qualified object, that is as if the state and a functions that use it will never return the same value. There is no standard interface that could change that state, but implementations may provide extensions that do so.

— **environ** is a placeholder for the overall shared environment list as accessed by the function **getenv** as of `<stdlib.h>` (7.22.5.6) or other implementation-specific functions that access the environment list. There is no standard interface that could change that state, but implementations may provide extensions that do so.

— **malloc** is a placeholder for the thread-local state of the storage management functions in `<stdlib.h>` (7.22.4) or other implementation-specific functions that allocate or deallocate storage.

— **atexit** and **at_quick_exit** are placeholders for the global state of the **atexit** and **at_quick_exit** handlers as in `<stdlib.h>` (7.22.4), respectively.

— **locale** is a placeholder for the overall shared locale as accessed by the function **setlocale** and **localeconv** from `<locale.h>` (7.11) and other locale-dependent functions.

— The following identifiers are used as placeholders for the static state that is used by the library functions with the same name.

`c16rtomb`	`mbrtoc16`	`mbsrtowcs`	`wcsrtombs`
`c32rtomb`	`mbrtoc32`	`mbtowc`	`wctomb`
`mbrlen`	`mbrtowc`	`wcrtomb`	

Additionally, function names may such as **`tmpnam`** of functions (library or application specific) may be used. The meaning is that under some circumstances the function may return a pointer to a **`static`** state, and thus that other functions may internally use that same state or use the same object for their return.

12 The **`core::reentrant`** attribute is the most restricted and describes those functions and lambda expressions that can be use as or by signal handlers.

13 The **`core::concurrent`** attribute describes a quite restricted set of functions or lambdas, too. Such functions and lambdas that may be robustly executed within different threads of execution without race conditions. Often they can even be performed once and for all at compile time if the arguments are constant expressions, and therefore this attribute also forms a main ingreedient for the formulation of the properties of the **`constexpr`** specifier for function definitions and lambda expressions.

14 The **`core::reinterpret`** attribute enforces compile time checks at the function call boundary, such that one hand, more properties such as qualifications of parameters, or array bounds can be propagated into the called function, and that on the other hand the function has relaxed constraints concerning type based aliasing.

15 The **`core::alias`** and **`core::noalias`** attributes have a wider application than as core function attributes. They have been described in 6.7.15.3.

6.7.15.4.1 The `FUNCTION_ATTRIBUTE` pragma

Syntax

1
`# pragma` `CORE FUNCTION_ATTRIBUTE` *attribute*
`# pragma` `CORE FUNCTION_ATTRIBUTE` *attribute-token on-off-switch*

Constraints

2 The *attribute* and *attribute-token* shall refer to a core function attribute.

Description

3 In the first form the attribute, including a possible attribute argument clause as described in the following clauses, is applied to function declarations and lambda expressions until one of the following conditions is encountered.

4 If the `FUNCTION_ATTRIBUTE` pragma is applied in file scope, it applies at most until the end of the current source file. If it is applied in block scope, it applies at most until the end of the current block, where the behavior for the attribute switches back to what it was before entering the block. If intermittent, another `FUNCTION_ATTRIBUTE` pragma is met that uses the same attribute token, and if the new `FUNCTION_ATTRIBUTE` pragma is of the first form, the attribute is further applied but with the newly specified attribute argument clause; if it is of the second form, the attribute is not further applied.

5 The `FUNCTION_ATTRIBUTE` pragma is only applied to to function declarations or lambda expressions that are found in the same source file, not to those that are included via an **`#include`** directive.

6 If used in the second form, **`ON`** enables the attribute as if it given in the form *attribute-token*; **`OFF`** disables the attribute and **`DEFAULT`** switches the usage of the attribute to the default. If not stated otherwise, the default for an attribute is to be disabled.

7 **EXAMPLE** The following two pragmas could be adequate to annotate header and source files with numerical functions.

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced
#pragma CORE FUNCTION_ATTRIBUTE core::modifies(errno, fenv)
```

8 All function declarations and lambda expressions that are found in such a file are as if they were directly declared with attributes `unsequenced` and `modifies(errno, fenv)`. If the specified functions then have no pointer parameters and return type, the translator would be able to deduce that calls to these functions and lambdas can be moved as early as their arguments are available. It could also conclude that the only possible state change can be found are the return value, `errno` or the floating-point state, and that changes to that state only depend on the concrete arguments that have been passed into the call. So effectively, such calls could also be moved as late as their return value, potentially modified `errno` or floating-point status are used.

9 The effect of the pragmas ends at the end of the source file or when pragmas similar to the following are met:

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced OFF
#pragma CORE FUNCTION_ATTRIBUTE core::modifies OFF
```

6.7.15.4.2 The `core::noleak` attribute

Constraints

1 For a function definition or lambda expression that only calls other functions or lambdas that have the `core::noleak` attribute and that calls no allocator, the `core::noleak` attribute is implied.

Description

2 A *storage leak* is an allocated storage instance for which the application has not stored a pointer value to which it has access. The `core::noleak` attribute asserts that the annotated function or lambda does not leak any storage instances.

3 Any storage instance that is allocated during any call to a function or lambda annotated with the `core::noleak` attribute shall be deallocated before the end of the call, with the exception of an allocated return value, if the function or lambda is itself an allocator.

Recommended Practice

4 It is recommended that for functions or lambdas with a `core::noleak` attribute implementations diagnose any of the following situations if they may:

— Store of a pointer to a newly allocated storage instance such that the value could be accessible after the function call ended.

— Use of a pointer to a newly allocated storage instance as the argument to a function call that is not known not to store the value in henceforth accessible storage.

— Use of pointer to a newly allocated storage instance as the return value of the function when the return type of the function does not itself have the `core::noalias` attribute.

— Not using a pointer to a newly allocated storage instance as argument to a `core::free` or `core::realloc` parameter of a nested function call, unless it is the return value of the function.

— There is a possible control flow of the function execution such that the count of `core::noalias` return values is higher than the count of `core::free` arguments (plus the possible function return if it is itself an allocator).

6.7.15.4.3 The `core::address_independent` attribute

Constraints

1 If applied to a function definition or lambda expression, the function body shall not form an implicit or explicit conversion from or to a pointer type other than

— to add a qualification to the pointed-to type,

— to form a null pointer from a null pointer constant,

— to convert a `void*` pointer that it allocates to point to the effective type with which the allocated storage instance will be used,

— to convert an argument of a `core::free` or `core::realloc` parameter to `void*`,

— to be used as a pointer parameter of calls to **memcpy** or **memmove**, provided that source and target pointer types of these calls (before conversion) have compatible base types, and that the argument to the parameter `n` is an integer constant expression with a value that is a multiple of the size of the base type,

— to be used as a pointer parameter of a call to **memset** where the other arguments are integer constant expressions such that the argument to the parameter `c` has value 0 and the argument to `n` is the multiple of the size of the base type.

2 Additionally the function body shall not use any function or lambda that has not the **core**::**address_independent** attribute and it shall not

— access lvalues of union types or members thereof that have a member of pointer type,

— use the **fread** or **fwrite** functions,

— use any function of the **printf** and **scanf** families of functions with a format that is not a string literal, or with a string literal format that contains a `%p` conversion specifier,

— use the **totext** type generic macro, or any of the derived features (**strdup**, **strndup**, **fputs** or **puts**) to store the textual representation of a pointer value.

3 For a function definition that fulfills the above constraints and that does not access lvalues of character type the **core**::**address_independent** attribute is implied even if not formed explicitly.

Description

4 The **core**::**address_independent** attribute asserts that under all circumstances the function or lambda executes without specific knowledge of the use and properties of the address space of the execution. In particular, it will not use specific numerical or textual representations of any address, and thus its outcome will be independent of specific choices of the execution platform this concerning.

5 A function with the **core**::**address_independent** attribute shall not expose any storage instance or synthesize a pointer value.

6 **NOTE 1** In the case that a function fulfills the constraints but manipulates bytes through lvalues of character type, the **core**::**address_independent** attribute is not easily deduced automatically for a function definition or lambda expressions. In such a situation the fact that the function does not expose or synthesizes pointer values has to be asserted by other means.

7 **NOTE 2** The **core**::**address_independent** attribute also asserts that a call to such function will not jeopardize the aliasing analysis of the caller, because concrete information about addresses will not leak between different parts of the program state.

6.7.15.4.4 The core::modifies attribute

Constraints

1 The attribute argument clause shall be omitted, be empty or consist of an identifier list. Each identifier in the list shall appear at most once. Let $\{O_1, \ldots, O_n\}$ be the (possibly empty) set of identifiers in the list. $O_1, \ldots, O_n$ shall have internal or external linkage and shall be visible in the scope of declaration (for functions) or evaluation (for lambdas), or shall be C library channels.

2 If in the same translation unit there are several declarations of the same function with the **core**::**modifies** attribute , they shall specify the same set $\{O_1, \ldots, O_n\}$ of identifiers, and these identifiers shall refer to the same object or channel. For any evaluation of the same lambda value, the identifiers $\{O_1, \ldots, O_n\}$ as in the lambda expression from which it originated shall be visible, and these identifiers shall refer to the same object or channel.

3 If the attribute is applied to the definition of a function or to a lambda expression, additional constraints apply. Let $\{L_1, \ldots, L_k\}$ be the complete set of identifiers of objects that are defined with static or thread-local storage duration in the function body.

— Within the function body, the type of identifiers of objects (including C library channels) that have internal or external linkage shall gain **const**-qualification, unless they appear in the set $\{O_1, \ldots, O_n, L_1, \ldots, L_k\}$.

— Any lambda expression, lambda value, or function specifier that is evaluated within the function body, shall also have the **modifies** attribute. For such a function the set of identifiers in their attribute shall be a subset of $\{O_1, \ldots, O_n\}$. For such a lambda expression or value the set of identifiers in their attribute shall be a subset of $\{O_1, \ldots, O_n, L_1, \ldots, L_k\}$.[226)]

4 If the **core**::**modifies** attribute is not explicitly applied to the definition of a function or to a lambda expression but there are $\{O_1, \ldots, O_n\}$ and $\{L_1, \ldots, L_k\}$ such that the above constraints are verified, and where the set $\{O_1, \ldots, O_n\}$ is minimal with that property, the attribute **modifies**($O_1, \ldots, O_n$) is implied.

Description

5 The **core**::**modifies** attribute indicates that only those identifiers of objects with internal or external linkage that are listed may be used directly or indirectly to modify their objects and, equally, that the C library channels used to modify the execution state must be listed as well.

6 If several translation units are linked into the program and have declarations of the same identifier as a function with external linkage and with the **core**::**modifies** attribute , they shall specify the same set $\{O_1, \ldots, O_n\}$ of identifiers, and these identifiers shall have external linkage or shall correspond to C library channels.

7 **EXAMPLE** Consider the following function that returns a freshly allocated string with the current date.

```
#include <time.h>
#include <stdlib.h>
char const* date_alloc(void) {
    char* ret = malloc(26);
    if (ret) ctime_r(time(nullptr), ret);
    return ret;
}
```

This function doesn't use any global symbols other than the library function it invokes. Therefore it implicitly has a **core**::**modifies** attribute that just accumulates all the channels that are modified by these. There is only such function, namely **malloc** that modifies global state, namely the state of the storage allocation library. Thus an improved declaration that reflects that knowledge would look as follows.

```
char const* date_alloc(void) [[core::modifies(malloc)]];
```

6.7.15.4.5 The core::evaluates attribute

Constraints

1 The attribute argument clause shall be omitted, be empty or consist of an identifier list. Each identifier in the list shall appear at most once. Let $\{I_1, \ldots, I_m\}$ be the (possibly empty) set of identifiers in the list. $I_1, \ldots, I_m$ shall have internal or external linkage, shall not have an atomic type or be **volatile** qualified,[227)] and shall be visible in the scope of declaration (for functions) or evaluation (for lambdas), or shall be C library channels.

2 If in the same translation unit there are several declarations of the same function with the **core**::**evaluates** attribute , they shall specify the same set $\{I_1, \ldots, I_m\}$ of identifiers, and these identifiers shall refer to the same object or channel. For any evaluation of the same lambda value, the identifiers $\{I_1, \ldots, I_m\}$ as in the lambda expression from which it originated shall be visible, and these identifiers shall refer to the same object or channel.

3 If the attribute is applied to the definition of a function or to a lambda expression, additional constraints apply. Let $\{L_1, \ldots, L_k\}$ be the complete set of identifiers of objects that are defined with static or thread-local storage duration in the function body, and let $\{O_1, \ldots, O_n\}$ be the identifiers that are listed in a **modifies** attribute, if any.

226) Because any function or lambda that are to be called must be evaluated first, this means that effectively direct calls can only be issued from the function body that have the same or more restricted **modifies** constraints for the evaluation of static or thread-local identifiers.

227) Atomic or volatile read accesses may change the visible state, at least temporarily. Programs that want to use such variables in this framework have to specify them in the list of a **modifies** attribute.

— The function body shall not evaluate identifiers of objects of internal or external linkage (and amoung those C library channels) other than those identified by $\{I_1, \ldots, I_m, O_1, \ldots, O_n, L_1, \ldots, L_k\}$.

— Within the function body, the type of identifiers of objects (including C library channels) in the set $\{I_1, \ldots, I_m\} \setminus \{O_1, \ldots, O_n\}$ shall gain **const**-qualification.

— If an identifier with function or lambda type is evaluated other than as the designator in a function call, it shall appear in $\{I_1, \ldots, I_m\}$, or, for lambdas, shall have been formed in the function body. Additionally, any lambda expression, lambda value, or function specifier that is evaluated within the function body, shall also have the **evaluates** attribute. For such a function the set of identifiers in their attribute shall be a subset of $\{I_1, \ldots, I_m, O_1, \ldots, O_n\}$. For such a lambda expression or lambda value the set of identifiers in their attribute shall be a subset of $\{I_1, \ldots, I_m, O_1, \ldots, O_n, L_1, \ldots, L_k\}$.[228)]

4 If the **core::evaluates** attribute is not explicitly applied to the definition of a function or to a lambda expression but there are $\{I_1, \ldots, I_m\}$, $\{O_1, \ldots, O_n\}$ and $\{L_1, \ldots, L_k\}$ such that the above constraints are verified, and where the set $\{I_1, \ldots, I_m\}$ is minimal with that property, the attribute **evaluates**($I_1, \ldots, I_m$) is implied.

Description

5 The **core::evaluates** attribute indicates that only those identifiers corresponding to functions or objects with internal or external linkage that are listed and those that are additionally found in a **modifies** attribute may be evaluated in the function body. Equally, it indicates that only the listed C library channels may be used to inspect other state of the execution that is defined by any of the library clauses. Additionally it enforces that those identifiers are not used to modify the underlying objects or channels, unless they are also listed in the **modifies** attribute.

6 If several translation units are linked into the program and have declarations of the same identifier as a function with external linkage and with the **core::evaluates** attribute , they shall specify the same set $\{I_1, \ldots, I_m\}$ of identifiers, and these identifiers shall have external linkage or shall be C library channels.

7 **EXAMPLE** For the example function `date_alloc` above, **time** and **ctime_r** evaluate global channels of the C library, namely both access the **time** channel, and **ctime_r** additionally the **locale** channel.

```
    char const* date_alloc(void)
          [[core::modifies(malloc), core::evaluates(locale, time)]];
```

6.7.15.4.6 The core::stateless attribute

Constraints

1 If the attribute is applied to the definition of a function or to a lambda expression, any objects of static or thread-local storage duration that are defined in the function body shall be **const**-qualified and not **volatile**-qualified.

2 For any function definition or lambda expression that fulfills the above constraints the **core::stateless** attribute is implied.

Description

3 A function that could be declared with the **core::stateless** attribute is called *stateless*.

4 **EXAMPLE** The example function `date_alloc` above declares no **static** variables, it also implicitly has the **core::stateless** attribute .

```
    char const* date_alloc(void)
          [[core::modifies(malloc), core::evaluates(locale, time)]]
          [[core::stateless]];
```

[228)]Because any function or lambda that are to be called must be evaluated first, this means that effectively direct calls can only be issued from the function body that have the same or more restricted **evaluates** constraints for the evaluation of static or thread-local identifiers.

6.7.15.4.7 The `core::state_invariant` attribute

Constraints

1 If not given explicitly, **`core::stateless`** and **`core::address_independent`** attributes are implied and the corresponding constraints apply. If no **`core::evaluates`** attribute is given explicitly, it is implied with an empty argument list and the corresponding constraints apply.

2 If the attribute is applied to the definition of a function or to a lambda expression, additional constraints apply. For any pointer or lambda value that is indirectly reachable from within the function body through the return value of a function call or through identifiers in **`evaluates`** or **`modifies`** attributes, in the parameter list or in captures, if any:

— If it has lambda type or pointer to function type it shall not be evaluated.

— If it has pointer to object type, it shall not be used with a unary $*$ operator or as left operand of a $\rightarrow$ operator such that the resulting lvalue is evaluated.[229)]

3 For a function definition or lambda expression that fulfills the above constraints, that is stateless, that fulfills the constraints of the **`core::address_independent`** and the **`core::evaluates`** attributes as specified, and that does not access lvalues of character type, the **`core::state_invariant`** attribute is implied even if not formed explicitly.

Description

4 The **`core::state_invariant`** attribute constrains the use of state of the execution to those objects that are explicitly identified in the control flow, that is that are the return values of function calls, that are parameters or captures, that appear in **`evaluates`** or **`modifies`** lists, or, if any of those has pointer type, that are accessible by directly dereferencing such pointers. In partcular, if it has no **`modifies`** and **`evaluates`** lists and none of the called functions or of the parameters or captures have lambda or pointer type, no state other than values of function calls, parameters and captures can be used by the function or lambda.

6.7.15.4.8 The `core::state_conserving` attribute

Constraints

1 If not given explicitly, **`core::stateless`**, **`core::noleak`** and **`core::address_independent`** attributes are implied and the corresponding constraints apply. If no **`core::modifies`** attribute is given explicitly, it is implied with an empty argument list and the corresponding constraints apply.

2 If the attribute is applied to the definition of a function or to a lambda expression, additional constraints apply. For any pointer or lambda value that is indirectly reachable from within the function body through the return value of a function call or through identifiers in **`evaluates`** or **`modifies`** attributes, in the parameter list or in captures, if any:

— If it has lambda type or pointer to function type it shall not be used as a function designator in a function call.

— If it has pointer to object type, the pointed-to type shall gain a **`const`** qualification.

3 For a function definition or lambda expression that fulfills the above constraints, that is stateless, that fulfills the constraints of the **`core::address_independent`** and the **`core::modifies`** attributes as specified, and that does not access lvalues of character type, the **`core::state_conserving`** attribute is implied even if not formed explicitly.

Description

4 The **`core::state_conserving`** attribute constrains the modification of state of the execution to those objects that are explicitly identified in the control flow, that is that are the return values of function calls, that are parameters or captures, that appear in **`evaluates`** or **`modifies`** lists, or, if any

[229)] This means that pointer to VM types are even prohibited to appear in **`sizeof`** expressions.

of those has pointer type, that are accessible by directly dereferencing such pointers. In partcular, if it has no **modifies** list and none of the function calls, parameters or captures have lambda or pointer type, no state change can be effected by the function or lambda other than through its return type.

6.7.15.4.9 The `core::state_transparent` attribute

Constraints

1 If not given explicitly, **`state_invariant`** and **`state_conserving`** attributes are implied and the corresponding constraints apply.

Description

2 The **`core::state_transparent`** attribute restricts all access to state of the execution to those objects that are either returns of functions, parameters or captures, or that appear in **`evaluates`** or **`modifies`** lists, or, if any of those has pointer type, that are accessible by directly dereferencing such pointers. In partcular, if it has no **`evaluates`** and **`modifies`** lists and none of the function calls, parameters or captures have lambda or pointer type, the effects of a call to the lambda or function are the value that is returned, if any, and the value is deterministically determined by the values of the arguments and captures.

6.7.15.4.10 The `core::idempotent` attribute

Constraints

1 The **`core::stateless`** and **`core::noleak`** attributes are implied and the corresponding constraints apply.

Description

2 An evaluation E is *idempotent* if it can be replaced by the evaluation `(`E`,`E`)` without changing the observable state of any execution. A function designator or lambda value identified by an identifier `f` is *idempotent* , if the evaluation `r = f(`$a_1,\ldots,a_n$`)` is idempotent, where the list $a_1,\ldots,a_n$ are **`const`** qualified variables that may range over the whole admissible set of function arguments (which may be empty), and where `r` is a variable with the non-**`void`** return type of the function or lambda. Analogously, `f` with a return type of **`void`** is *idempotent* if `f(`$a_1,\ldots,a_n$`)` is idempotent with $a_1,\ldots,a_n$ as above.

3 A function definition that has the **`idempotent`** attribute shall be such that the function designator is idempotent. A lambda expression that has the **`idempotent`** attribute shall be such that the lambda value, when assigned to a variable of lambda type, is idempotent.

4 If a function or lambda has the **`state_conserving`** attribute, and the identifier list of the **`modifies`** attribute is empty the **`core::idempotent`** attribute is implied.

6.7.15.4.11 The `core::independent` attribute

Constraints

1 The **`core::stateless`** and **`core::noleak`** attributes are implied and the corresponding constraints apply.

Description

2 A function or lambda call is *independent* , if all lvalue conversions that are effected during the call, inclusive calls into other functions or lambdas, refer to objects that are

— function parameters or captures of the called function or lambda expression,

— objects with static or thread-local storage duration that are **`const`** but not **`volatile`** qualified,

— objects for which the definition is met during the call, or

— objects that are allocated during the call.

3 A function definition is *independent* , if all function calls with the function designator and valid parameters are independent. A lambda expression is *independent* , if all function calls with

— a lambda value that is deduced from the expression, or

— with a function pointer that is converted from such a deduced lambda value

and valid parameters are independent.

4 A function definition that has the **core::independent** attribute shall be such that the function designator is independent. A lambda expression that has the **core::independent** attribute shall be such that the lambda value, when assigned to a variable of lambda type, is independent.

5 If a function or lambda has the **core::state_invariant** attribute, and the identifier list of the **evaluates** attribute is empty or has only identifiers of objects that are **const** but not **volatile** qualified, either because it was implied or given explicitly, the **core::independent** attribute is implied.

6.7.15.4.12 The core::unsequenced attribute

Constraints

1 The **core::independent** and **core::idempotent** attributes are implied and the corresponding constraints apply.

2 For any function declaration or lambda expression that has the **core::independent** and **core::idempotent** attributes the **core::unsequenced** attribute is implied.

3 For a function definition or for a lambda expression the **core::unsequenced** attribute is implied even if not formed explicitly, if the following conditions hold:

— It has the **core::stateless**, **core::noleak**, **core::address_independent** attributes.

— It has the **core::evaluates** attribute with an identifier list that is empty or that contains only names of objects that are **const** qualified and not **volatile** qualified.

— It has the **core::modifies** attribute with an identifier list that is empty.

— It only calls functions directly, that is where the function designator is converted to the function pointer within the calling expression.

— It only calls functions or lambdas that have the **core::unsequenced** attribute .

— It does not access lvalues of character type.

— No pointer or lambda value that is indirectly reachable from within the function body through the return value of a function call or through identifiers in **core::evaluates** or **core::modifies** attributes, in the parameter list or in captures, is evaluated.

Description

4 The **core::unsequenced** attribute indicates that a call to a function or lambda can be effected as soon as the values of its parameters, captures or **core::evaluates** and **core::modifies** channels (and objects to which they point) have been determined, and that it can be effected as late as any of its return value, modified pointed-to parameters, captures or **core::modifies** channels are accessed.

5 **NOTE** The **core::unsequenced** attribute asserts strong properties for the annotated function or lambda, in particular it accumulates the attributes **core::stateless**, **core::noleak**, **core::state_invariant**, and **core::state_conserving**. Thereby, calls to such functions or lambdas are natural candidates for optimization techniques such as common subexpression elimination, local memoization or lazy evaluation.

6.7.15.4.13 The core::concurrent attribute

Constraints

1 For a function definition or for a lambda expression annotated with the **core::concurrent** attribute the following contraints shall hold:

— If it does not have it, the `core::state_transparent` attribute shall be implied and the corresponding constraints shall apply.

— If it does not have it, a `core::evaluates` attribute with empty identifier list shall be implied. Otherwise, the identifier list shall be empty or it shall only contain names of objects that are `const` qualified and not `volatile` qualified, or that have thread local storage duration.

— If it does not have it, a `core::modifies` attribute with an empty identifier list shall be implied. Otherwise, the identifier list shall be empty or it shall only contain names of objects that have thread local storage duration.

— It shall only call functions directly, that is where the function designator is converted to the function pointer within the calling expression.

— It shall only call functions or lambdas that have the `core::concurrent` attribute .

— No pointer or lambda value that is indirectly reachable from within the function body through the return value of a function call or through identifiers in `core::evaluates` or `core::modifies` attributes, in the parameter list or in captures, shall be evaluated.

2 For a function definition or for a lambda expression that fulfills the above constraints, the `core::concurrent` attribute is implied.

Description

3 The `core::concurrent` attribute indicates that calls to the function or lambda can be effected concurrently, as soon as the values of its parameters, captures or `core::evaluates` and `core::modifies` channels (and objects to which they point) have been determined, and that it can be effected as late as any of its return value, modified pointed-to parameters, captures or `core::modifies` channels are accessed. Such a concurrent execution between calls is race-free, as long as all pointer parameters or captures of the calls refer to mutually disjoint sets of objects.

4 **NOTE** The `core::concurrent` attribute asserts strong properties for the annotated function or lambda, in particular it accumulates the attributes `core::stateless`, `core::noleak`, `core::state_invariant`, and `core::state_conserving`. The required properties are weaker than for the `core::unsequenced` attribute, in particular such a function may be neither idempotent nor independent, as long as the state changes that violate these properties are thread local. In particular:

— As long as it has no leak, it may allocate objects and thus change the `malloc` channel.

— It may access the `errno`, `math_errhandling` and `fenv` channels.

6.7.15.4.14 The `core::reentrant` attribute

Constraints

1 Let `f` be a function definition or lambda expression that fulfills all the following properties:

— It has the `state_transparent` attribute.

— The channels of the `modifies` and `evaluates` attributes have a lock-free atomic type, `atomic_flag` or `sig_atomic_t`.

— If a parameter or capture and has a pointer type it is a pointer to a lock-free atomic type, `atomic_flag` or `sig_atomic_t`.

— If a channel or a pointed-to parameter or capture has the type `sig_atomic_t` it shall not be `const`-qualified and only be used as the left operand of an assignment.

— The return type is not a pointer type.

— None of its parameters, captures and return type have a lambda type.

— In addition to other core function constraints for called functions, all functions and lambdas that are called by `f` have the **`reentrant`** attribute[230] or they are a type-generic operation on a lock-free atomic type (7.17) other than **`atomic_init`**.

Then the **`core`**:: **`reentrant`** attribute shall be implied for `f`.

Description

2 A function or lambda is *reentrant* , if two or more calls to the function can be active in the same thread of execution without being sequenced. Reentrant functions can be called by signal handlers, or, if not used in the context of a signal handler, they can be traversed by a call to **`longjmp`** without jeopardizing the execution.

3 A function declaration (including a definition) that has the **`reentrant`** attribute shall designate a reentrant function. A lambda expression that has the **`reentrant`** attribute shall be reentrant.

4 As an exception to the general constraints for core function attributes, a function definition or lambda expression that has the **`core`**:: **`reentrant`** attribute

— may call other functions or lambdas without the **`core`**:: **`reentrant`** attribute ,

— may have declarations that have not the **`core`**:: **`reentrant`** attribute .[231]

5 **NOTE** To prove reentrancy, all possible interleaved executions of two or more calls of the same function have to be considered. Therefore, this property is generally difficult to assert automatically by the translator. The implied application of the **`core`**:: **`reentrant`** attribute as given int the constraints only proves reentrancy for a subset of the functions and lambdas that are effectively reentrant. If needed, applications would have to prove that property by other means and to annotate functions definitions or lambda expressions explicitly with the **`core`**:: **`reentrant`** attribute .

6 **EXAMPLE** Consider the following example code that uses converted lambda values as a signal handlers:

```
#include <signal.h>
extern sig_atomic_t bad;
extern atomic_type(size_t) counter;

signal(SIGSEGV,
  [](int sig) {
    bad = 1;
  }
);

signal(SIGINT,
  [](int sig) { // diagnosis if size_t is not lock-free
    ++counter;
  }
);
```

The only identifiers that are accessed by the lambda expressions are `bad` and `counter`, respectively, so **`core`**:: **`evaluates`**`()` and **`core`**:: **`modifies`**`(bad)` (or **`core`**:: **`modifies`**`(counter)`) attributes are implied. Additionally, the lambdas define no object of static or thread-local storage duration, so **`core`**:: **`stateless`** is implied for both, and since no other operations than the assignment (or increment) are formed the **`core`**:: **`state_transparent`** attribute follows, too. To assess the **`core`**:: **`reentrant`** attribute, the variables have to be checked if they fulfill the type constraints. The first lambda unconditionally fulfills the constraints, whereas the second only fulfills them if and only if **`size_t`** is lock-free. If not, as a consequence the implementation may issue a diagnosis to warn that the second lambda is not reentrant.

Forward references: signal handling (7.14), atomics (7.17).

6.7.15.4.15 The `core`:: `reinterpret` attribute

[230] This is needed to ensure that the called functions or lambdas do not have non-atomic pointer parameters, either. Because the function has no channels that have function pointer type, a possible conversion of a function or a lambda to a function pointer must be visible within the function body. So the **`core`**:: **`reentrant`** attribute (and thus implicitly the property being reentrant) can be verified at translation time, if all these properties are assembled.

[231] Any declaration that is not a definition of a function with a **`core`**:: **`reentrant`** attribute , even implied, still forces the definition of the function to have the **`core`**:: **`reentrant`** attribute as well.

Constraints

1 The `core::reinterpret` attribute shall not be applied to a function declarator or lambda expression with variable argument list. If a function definition has the `core::reinterpret` attribute, all declarations in the same translation unit shall be token equivalent.

2 If a declaration of object A of type T is visible and if A is pointed-to by an argument passed to a pointer parameter of type $S*$ of a function that has the `core::reinterpret` attribute, A shall be suitably aligned for S and T shall be representable by S.

3 If object A acquired the effective type T prior to the call within in the calling function or by being the pointed-to argument of the calling function itself, and if A is pointed-to by an argument passed to a pointer parameter of type $S*$ of a function that has the `core::reinterpret` attribute, T shall be representable by S.[232] If A is itself a parameter of a function with the `core::reinterpret` attribute and if T is specified with array notation, the type for this constraint is the type before it is rewritten to a pointer type.

Description

4 The `core::reinterpret` attribute loosens type constrains for functions and lambdas that have parameters of pointer to object type, by ensuring compatible qualifications, sizes and representations between arguments and parameters all the same. For the whole duration of the call to such a function, the objects that are pointed-to by such parameters shall have the effective type indicated by the prototype, and this regardless on how the object was originally declared, if it was declared, or how it possibly received an effective type prior to the call. If prior to the call such an object had an effective type in the context of the caller, it retains that effective type; if it had no effective type it gains the type as in the function call.

5 If object A has effective type T and is pointed-to by an argument passed to a pointer parameter of type $S*$ of a function that has the `core::reinterpret` attribute, A shall be suitably aligned for S and T shall be representable by S.

6 If several translation units are linked into the program and have declarations of the same identifier as a function with external linkage such that the function definition has the `core::reinterpret` attribute, then all declarations shall have the `core::reinterpret` attribute. If the function is called from a different translation unit than its definition, the value expressions that are present in array bounds for parameters, shall evaluate to the same values in the context of the caller as in the context of the definition; all qualifications of corresponding function parameters shall be the same.

7 The default value for the `CORE FUNCTION_ATTRIBUTE` pragma as of 6.7.15.4.1 for the `core::reinterpret` attribute is implementation-defined.

8 **EXAMPLE 1** Consider the following function that initializes a `uint16_t` vector. Here it is assumed that alignment constraints for `uint16_t` are at most as strict as for `uint32_t`.

```
void init(size_t len, uint16_t a[len]) [[core::reinterpret]] {
    for (size_t i = 0; i < len; ++i) a[i] = 0;
}
    ...
    size_t n = ...
    ...
    uint32_t vec32[n];
    size_t m = sizeof(vec32)/sizeof(uint16_t);
    init(m, (uint16_t(*)[m])&vec32);

    alignas(uint16_t) void buf16[sizeof(uint16_t[n])];
    float (*vec16)[n] = buffer;
    init(n, *vec16);
```

9 The first call to `init` would be valid by itself, because the passed pointer has the correct type, namely `uint16_t*`.

[232] Note that here no constraint can be enforced for the alignment of A. Nevertheless alignment requirements are implied below and may lead to undefined behavior if not respected.

Nevertheless, if the function would not have the **core**::**reinterpret** attribute the effective type rule would be violated by the access to the elements, and thus the behavior of the call would then be undefined.

10 With the **core**::**reinterpret** attribute that aspect of the function call is avoided. During the call of the function the vector can be accessed as if it were defined as having type **uint16_t**. In the context of the caller, the effective type still remains **uint32_t**`[len]`. Additionally, a check for the correctness of the call can be performed at translation time: an array of type **uint16_t** can be represented by an array of type **uint32_t** (because these types have no padding) and the argument passed to the `len` parameter is correct.

11 The initialization of `vec16` by `buffer` does not change the effective type of the pointed-to object, because up to that point nothing has been written into it. The call to `init` then changes this; not only is the effective type within the function call **uint16_t**`[len]` but it remains so after the return from the function.

12 **EXAMPLE 2** Consider the following function for vector addition that is used inside several functions that add matrices

```
void addup(size_t len, float a[len], decltype(a) b) [[core::reinterpret]] {
      for (size_t i = 0; i < len; ++i) a[i] += b[i];
}

// valid, renames len
void addup(size_t n, float a[n], float b[n]) [[core::reinterpret]];
// invalid, omits the attribute
void addup(size_t k, float a[k], decltype(a) b);
// invalid, omits len
void addup(size_t, float a[], float b[]) [[core::reinterpret]];

void matadd0(size_t n, size_t m, complex_type(float) A[n][m], decltype(A) B){
      addup(2*n*m, A, B);                             // invalid, constraint violation
}

void matadd1(size_t n, size_t m, complex_type(float) A[n][m], decltype(A) B){
      addup(2*n*m, (void*)&A[0][0], (void*)&B[0][0]); // valid
}

void matadd2(size_t n, size_t m, int A[n][m], decltype(A) B) [[core::reinterpret]] {
      addup(n*m, (void*)&A[0][0], (void*)&B[0][0]);   // invalid, constraint violation
}

void matadd3(size_t n, size_t m, float A[n-1][m], decltype(A) B) [[core::reinterpret]] {
      addup(n*m, (void*)&A[0][0], (void*)&B[0][0]);   // invalid, constraint violation
}
```

Here, `matadd0` has a constraint violation because the types of the arguments `A` and `B` are not compatible with the parameters `a` and `b`, respectively.

13 For `matadd1` the problem of the compatible types is resolved (a bit rudely), but now the call to the function with the **core**::**reinterpret** attribute is presented with a pointer to an object (a complex **float** matrix) that is able to represent the parameter type, a **float** vector. Thus the call is valid, and the translator is even able to detect this.

14 For `matadd2` the problem of the compatible types is resolved the same, but now the call to the function with the **core**::**reinterpret** attribute is presented with a pointer to an object (an **int** matrix) that cannot represent the parameter type, a **float** vector, even though for many platforms **int** and **float** may have the same size and alignment. Thus the call is invalid. Because `matadd2` also has the **core**::**reinterpret** attribute , the translator is able to deduce that the effective type of the pointed-to objects here in all cases is to be assumed to be **int**`[n][m]` so it has to detect this and to issue a diagnostic.

15 `matadd3` shows another constraint violation that might arise. The size of the pointed-to object *A* is deduced *before* the parameter is rewritten, so the translator may assume an object of size **sizeof**`(`**float**`[n-1][m])` that is `(n-1)*m*`**sizeof**`(`**float**`)`. On the other hand, `addup` is known to expect an argument of size **sizeof**`(`**float**`[len])` that is `n*m*`**sizeof**`(`**float**`)`. Thus the parameter size is larger than the argument size and thus the parameter can not be represented by the argument. Effectively, the execution of `addup` would lead to an out-of-bounds access, which, by the help of the **core**::**reinterpret** attribute , is detected at translation time.

16 **EXAMPLE 3** Consider a similar example, but this time with generic lambdas.

```
#define λ0 [](size_t len, auto a[len], decltype(a) b) [[core::reinterpret]] { \
   for (size_t i = 0; i < len; ++i) a[i] += b[i];                         \
}

#define λ1 [](size_t n, size_t m, auto A[n][m], decltype(A) B) [[core::reinterpret]] { \
```

```
  λ0(n*sizeof(A[0])/sizeof(real_type(A[0][0])),                         \
     (real_type(+A[0][0])*)&A[0][0],                                    \
     (real_type(+B[0][0])*)&B[0][0]);    /* valid */                    \
}
```

First, it is easy to see that λ_0 is valid for any arithmetic type. λ_1 too is valid for any arithmetic type: if the type is a real type, the casts in the argument expression for the call to λ_0 are a no-ops. If the type is a complex type, the cast is a cast to the real type, which has the same representation. The **core::reinterpret** attribute for λ_1 enforces an interpretation of the pointed to parameters as having the effective type as it is inferred for the specific call or conversion of the lambda value λ_1.

6.8 Statements and blocks

Syntax

1 *statement:*

> *labeled-statement*
> *expression-statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ *compound-statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ *selection-statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ *iteration-statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ *jump-statement*

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence. The optional attribute specifier sequence appertains to the respective statement.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 **NOTE** Each of the following is a full expression:

- a full declarator for a variably modified type,
- an initializer that is not part of a compound literal,
- the expression in an expression statement,
- the controlling expression of a selection statement (**`if`** or **`switch`**),
- the controlling expression of a **`while`** or **`do`** statement,
- each of the (optional) expressions of a **`for`** statement,
- the (optional) expression in a **`return`** statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *labeled-statement:*

> *attribute-specifier-sequence*$_{\text{opt}}$ *identifier* **`:`** *statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ **`case`** *constant-expression* **`:`** *statement*
> *attribute-specifier-sequence*$_{\text{opt}}$ **`default`** **`:`** *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement ~~.~~ that is associated with the same function body as the statement to which the label is attached.[233] Further constraints on such labels are discussed under the **`switch`** statement.

[233] Thus, a label that appears within a lambda expression may only be associated to a switch statement within the body of the lambda.

3 Label names shall be unique within a function.

Semantics

4 Any statement may be preceded by a prefix that declares an identifier as a label name. The optional attribute specifier sequence appertains to the label. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Forward references: the **goto** statement (6.8.6.1), the **switch** statement (6.8.4.2).

6.8.2 Compound statement

Syntax

1 *compound-statement:*
  { *block-item-list*$_{\text{opt}}$ **}**
block-item-list:
  block-item
  block-item-list block-item
block-item:
  declaration
  statement

Semantics

2 A *compound statement* is a block.

6.8.3 Expression and null statements

Syntax

1 *expression-statement:*
  expression$_{\text{opt}}$ **;**
  attribute-specifier-sequence expression **;**

Semantics

2 The attribute specifier sequence appertains to the expression. The expression in an expression statement is evaluated as a void expression for its side effects.[234)]

3 A *null statement* (consisting of just a semicolon) performs no operations.

4 **EXAMPLE 1** If a function call is evaluated as an expression statement for its side effects only, the discarding of its value can be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

5 **EXAMPLE 2** In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
while (*s++ ≠ '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

6 **EXAMPLE 3** A null statement can also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
        /* ... */
        while (loop2) {
                /* ... */
                if (want_out)
                        goto end_loop1;
```

234)Such as assignments, and function calls which have side effects.

```
            /* ... */
        }
        /* ... */
    end_loop1:;
    }
```

Forward references: iteration statements (6.8.5).

6.8.4 Selection statements

Syntax

1 *selection-statement:*
    if (~~*expression*) *statement*~~ *init-statement*$_{opt}$ *controlling-expression*) *consequent-body*
    if (~~*expression*) *statement* **else** *statement*~~ *init-statement*$_{opt}$ *controlling-expression*) *consequent-body* **else** *alternative-body*
    switch (~~*expression*) *statement*~~ *controlling-expression*) *switch-body*

init-statement:
    expression-statement
    declaration

consequent-body:
    statement

alternative-body:
    statement

switch-body:
    statement

Constraints

2 The declaration part of an init statement shall only declare identifiers for objects having storage class **auto**.[235)]

Semantics

3 A selection statement selects among a set of statements depending on the value of a controlling expression.

4 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated ~~substatement~~ body is also a block whose scope is a strict subset of the scope of the selection statement.

5 **NOTE** The possibility of an init statement for the **if** statement is a construct that is currently only present in C++, not in C. We add it here, because is a comfortable tool to catch error return codes from C library functions and to continue execution conditionally on the return value. Implementations that want to target the common C/C++ core have to add this feature to comply with this specification.

6.8.4.1 The `if` statement

Constraints

1 The controlling expression of an **if** statement shall have scalar type.

2 If the consequent body or the alternative body are a compound statement, they shall not themselves contain declarations of the same identifiers as the declaration part of the init statement, if any.[236)]

Semantics

3 In both forms, the ~~first substatement~~ consequent body is executed if the expression ~~compares unequal to 0.~~ yields **true** when converted to **bool**. In the **else** form, the ~~second substatement~~

[235)] Thus, any such declaration names an object of automatic storage duration and is also a definition.

[236)] This provision only holds for the compound statement itself. Other blocks that are nested within that first compound statement may redeclare any identifier according to the scoping rules.

alternative body is executed if the expression ~~compares equal to 0. If the first substatement~~ yields **`false`**. If the consequent body is reached via a label, the ~~second substatement~~ alternative body is not executed.

4 An **`else`** is associated with the lexically nearest preceding **`if`** that is allowed by the syntax.

5 If the init statement is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire **`if`** statement, including the controlling expression; it is reached in the order of execution before the evaluation of the controlling expression. If the init statement is an expression, it is evaluated as a void expression before the evaluation of the controlling expression.[237)]

6.8.4.2 The `switch` statement

Constraints

1 The controlling expression of a **`switch`** statement shall have integer type.

2 If a **`switch`** statement has an associated **`case`** or **`default`** label within the scope of an identifier with a variably modified type, the entire **`switch`** statement shall be within the scope of that identifier.[238)]

3 The expression of each **`case`** label shall be an integer constant expression and no two of the **`case`** constant expressions in the same **`switch`** statement shall have the same value after conversion. There may be at most one **`default`** label in a **`switch`** statement. (Any enclosed **`switch`** statement may have a **`default`** label or **`case`** constant expressions with values that duplicate **`case`** constant expressions in the enclosing **`switch`** statement.)

Semantics

4 A **`switch`** statement causes control to jump to, into, or past the ~~statement that is the~~ switch body, depending on the value of a controlling expression, and on the presence of a **`default`** label and the values of any **`case`** labels on or in the switch body. A **`case`** or **`default`** label is accessible only within the closest enclosing **`switch`** statement.

5 The integer promotions are performed on the controlling expression. The constant expression in each **`case`** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **`case`** label. Otherwise, if there is a **`default`** label, control jumps to the labeled statement. If no converted **`case`** constant expression matches and there is no **`default`** label, no part of the switch body is executed.

Implementation limits

6 As discussed in 5.2.4.1, the implementation may limit the number of **`case`** values in a **`switch`** statement.

7 **EXAMPLE** In the artificial program fragment

```
switch (expr)
{
      int i = 4;
      f(i);
case 0:
      i = 17;
      /* falls through into default code */
default:
      printf("%d\n", i);
}
```

the object whose identifier is `i` exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **`printf`** function will access an indeterminate value. Similarly,

[237)] Thus, the init statement specifies an initialization for the **`if`** statement, possibly defining one or more variables; the controlling expression specifies an evaluation that determines which (or if any) of the bodies is executed and that may use (or just be) a variable that had been defined in the init statement.

[238)] That is, the declaration either precedes the **`switch`** statement, or it follows the last **`case`** or **`default`** label associated with the **`switch`** that is in the block containing the declaration.

the call to the function `f` cannot be reached.

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*

while (~~*expression*) *statement*~~ *controlling-expression*) *loop-body*
do ~~*statement* **while** (*expression*~~ *loop-body* **while** (*controlling-expression*) ;
for (~~*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*~~
~~**for** (*declaration expression*$_{opt}$; *expression*$_{opt}$) *statement*~~
init-statement controlling-expression$_{opt}$; *iteration-expression*) *loop-body*

loop-body:

statement

iteration-expression:

expression$_{opt}$

Constraints

2 The controlling expression of an iteration statement shall have scalar type.

3 ~~The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**.~~ If the loop body is a compound statement, it shall not itself contain a declaration of the same identifiers as the declaration part of the init statement, if any.[239)]

Semantics

4 An iteration statement causes ~~a statement called the~~ the loop body to be executed repeatedly until the controlling expression ~~compares equal to 0.~~ yields **false** when converted to **bool**. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[240)]

5 An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

6 An iteration statement may be assumed by the implementation to terminate if its controlling expression is not a constant expression,[241)] and none of the following operations are performed in its body, controlling expression or (in the case of a **for** statement) its ~~*expression-3*~~ iteration expression:[242)]

— input/output operations

— accessing a volatile object

— synchronization or atomic operations.

7 **NOTE 1** C and C++ differ in the scoping of variables that are declared in a declaration part of an init statement. C has less constraints and allows that shadowing identifiers are declared in the top level compound statement of the loop body. Applications that target the common C/C++ core should not rely on this feature and implementations conforming to this specification have to diagnose such declarations as being not portable.

8 **NOTE 2** C++ has additional possibilities to declare (and define) loop variables, namely the controlling expression of **for** and **while** loops may declare and initialize variables, such that the value of the controlling expression is the value of the variable. For C it is quite uncommon to consider that a declaration could be an expression, "have a value". Therefore we did not chose to add this feature to C. A **while** with such a feature can easily be emulated by a usual **for** loop:

```
while (int myControl = (complicated-expression)) {              // only C++
   // use ``myControl'' somehow
}
for (int myControl; (myControl = (complicated-expression)); ) { // same, C and C++
   // use ``myControl'' somehow
```

239)This provision only holds for the compound statement itself. Other blocks that are nested within that first compound statement may redeclare any identifier according to the scoping rules.

240)Code jumped over is not executed. In particular, the controlling expression of a **for** or **while** statement is not evaluated before entering the loop body, nor is the init statement of a **for** statement.

241)An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

242)This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.

```
}
```

Implementations that want to target the common C/C++ should diagnose the occurence of such "controlling expression variables" as to be non-portable.

6.8.5.1 The `while` statement

1 The evaluation of the controlling expression takes place before each execution of the loop body.

6.8.5.2 The `do` statement

1 The evaluation of the controlling expression takes place after each execution of the loop body.

6.8.5.3 The `for` statement

1 The ~~statement behaves as follows: The expression *expression-2* is the controlling expression that~~ controlling expression is evaluated before each execution of the loop body. The ~~expression *expression-3*~~ iteration expression is evaluated as a void expression after each execution of the loop body. If ~~*clause-1*~~ the init statement is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If ~~*clause-1*~~ the init statement is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[243)]

2 ~~Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2*~~ An omitted controlling expression is replaced by ~~a nonzero constant.~~**`true`**.[244)]

6.8.6 Jump statements

Syntax

1 *jump-statement:*

```
goto identifier ;
continue ;
break ;
return expression_opt ;
```

Constraints

2 No jump statement other than **`return`** shall have a target that is found in another function body.[245)]

Semantics

3 A jump statement causes an unconditional jump to another place.

6.8.6.1 The `goto` statement

Constraints

1 The identifier in a **`goto`** statement shall name a label located somewhere in the enclosing function body. A **`goto`** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier, or into or out of the scope of a lambda expression.

Semantics

2 A **`goto`** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

3 **EXAMPLE 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline

[243)] Thus, the init statement specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression specifies an evaluation made before each iteration, such that execution of the loop continues until the expression yields **`false`**; and the iteration expression specifies an operation (such as incrementing) that is performed after each iteration.

[244)] As the syntax implies, both, the init statement and the iteration expression can also essentially be empty. Effectively, a **`for`** statement of the form "**`for(;;)`** *loop-body*" performs an non-terminating loop.

[245)] Thus jump statements other than **`return`** may not jump between different functions or cross the boundaries of a lambda expression, that is, they may not jump into or out of the function body of a lambda. Other features such as signals (7.14) and long jumps (7.13) may delegate control to points of the program that do not fall under these constraints.

presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **continue** statements, for example.)

```
/* ... */
goto first_time;
for (;;) {
        // determine next operation
        /* ... */
        if (need to reinitialize) {
                // reinitialize-only code
                /* ... */
        first_time:
                // general initialization code
                /* ... */
                continue;
        }
        // handle other operations
        /* ... */
}
```

4 **EXAMPLE 2** A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
goto lab3;                  // invalid:  going INTO scope of VLA.
{
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4;          // valid:  going WITHIN scope of VLA.
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
}
goto lab4;                  // invalid:  going INTO scope of VLA.
```

6.8.6.2 The continue statement

Constraints

1 A **continue** statement shall appear only in or as a loop body ~~.~~ that is associated to the same function body.[246)]

Semantics

2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

`while (/* ... */) {`
`   /* ... */`
`   continue;`
`   /* ... */`
`contin:;`
`}`
	`do {`
	`   /* ... */`
	`   continue;`
	`   /* ... */`
	`contin:;`
	`} while (/* ... */);`
		`for (/* ... */) {`
		`   /* ... */`
		`   continue;`
		`   /* ... */`
		`contin:;`
		`}`

246) Thus a **continue** statement by itself may not be used to terminate the execution of the body of a lambda expresssion.

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto** `contin`;.[247)]

6.8.6.3 The `break` statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body ~~.~~ that is associated to the same function body.[248)]

Semantics

2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

6.8.6.4 The `return` statement

Constraints

1 ~~A **return** statement with an expressionshall not appear in a function whose return type is **void**~~ The expression, if any, shall not have an opaque type other than **void**. If the return type of the function is not **void** the type of the expression shall be such that it can be converted to the function return type as if by assignment. A **return** statement without an expression shall only appear in a function whose return type is **void**.

2 For a lambda expression or function that has an underspecified return type, all **return** statements shall provide expressions with a consistent type. That is, if any **return** statement has an expression that is not a **void** expression, all **return** statements shall have an expression with the same generic type; otherwise all **return** expressions shall have no expression or one that is a **void** expression.

Semantics

3 A **return** statement ~~terminates execution of the current function~~ is associated to the innermost function body in which appears. It evaluates the expression, if any, terminates the execution of that function body and returns control to its caller~~. A function~~ ; if it has an expression other than a **void** expression, the value of the expression is returned to the caller as the value of the function call expression. A function body may have any number of **return** statements.

4 ~~If a **return** statement with an expression is executed, the value of the expressionis returned~~ For a function that has a specified return type, if the function return type is **void**, the expression, if any, is evaluated as a **void** expression and control returns to the caller~~as the value of the function call expression. If~~ . Otherwise, if the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[249)]

5 For a lambda expression or function that has an underspecified return type, the return type is determined as soon as a **return** statement is met and is specified as the generic type of that expression, if any, or as **void** if there is no expression or if no **return** statement is found.

6 **EXAMPLE** In:

```
        struct s { double i; } f(void);
        union {
              struct {
                    int f1;
                    struct s f2;
              } u1;
              struct {
                    struct s f3;
                    int f4;
              } u2;
        } g;
```

[247)] Following the `contin`: label is a null statement.

[248)] Thus a **break** statement by itself may not be used terminate the execution of the body of a lambda expresssion.

[249)] The **return** statement is not an assignment. The overlap restriction of 6.5.17.1 does not apply to the case of function return. The representation of floating-point values can have wider range or precision than implied by the type; a cast can be used to remove this extra range and precision.

```
struct s f(void)
{
      return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

6.9 External definitions

Syntax

1 *translation-unit:*
: *external-declaration*
: *translation-unit external-declaration*

external-declaration:
: *function-definition*
: *declaration*

Constraints

2 ~~The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.~~

There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression ~~(other than as a part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant),~~that is evaluated,[250] there shall be exactly one external definition for the identifier in the translation unit.

Semantics

3 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes ~~storage~~ a storage instance to be reserved for an object or provides the body of a function named by the identifier is a definition.

4 An *external definition* is an external declaration that is also a definition of a function ~~(~~or an object other than an inline definition~~) or an object. If~~. Unless specified otherwise, if an identifier declared with external linkage is used in an expression ~~(other than as part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant)~~that is evaluated, somewhere in the entire program there shall be exactly one external definition for the identifier;[251] otherwise, there shall be no more than one.[252]

Recommended practice

5 C++ has looser rules than this for certain special cases. Even if the identifiers are evaluated, inline functions and objects are not required to provide an external definition. If needed, a program-wide unique address that stands for an external definition is provided. Also, **const**-qualification of objects may imply internal linkage in some cases, such that linker conflicts are avoided.

6 Applications that target the common C / C++ core should avoid such situations. In particular, they should provide an external definition for all functions and objects with external linkage, and they should augment the definitions of **const** qualified objects to be inline objects if possible. In many cases this has the advantage of promoting compile-time constant expressions of integer type to "integer constant expressions", and, by that, of transforming definitions of VLA (with a compile time known size) to ordinary arrays that can be initialized.

7 It is recommended that implementations diagnose these situations whenever they may, in particular when they encounter **const**-qualified objects that would qualify to be transformed into

[250] Several expressions that are only inspected for their type are not evaluated. This may or may not apply to dependent expressions in **generic_selection** primary expressions, the **decltype** specifier, the **sizeof** operator, and the **alignof** operator.

[251] Exempted from having an external definition are inline constants if they are only used in lvalue conversions.

[252] Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

inline constants and point to situations where this would avoid the definition of a VLA.

6.9.1 Function definitions

Syntax

1 *function-definition:*

attribute-specifier-sequence$_{\mathrm{opt}}$ *declaration-specifiers declarator function-body*

function-body:

compound-statement

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.

3 The return type of a function shall be `void` or a complete object type other than ~~array type~~an array type or an opaque type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either `extern` or `static`, possibly combined with `auto`.

5 ~~The declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting~~ If the parameter type list consists of a single parameter of type `void`, ~~in which case there shall not be~~ the parameter declarator shall not include an identifier.

6 An underspecified function definition shall contain at least one storage class specifier. The return type for such a function is determined as described for the `return` statement, 6.8.6.4.[253]

Semantics

7 The optional attribute specifier sequence in a function definition appertains to the function. If `auto` appears as a storage-class specifier it is ignored for the purpose of determining a storage class or linkage of the function. It then only indicates that the return type of the function may be inferred from `return` statements, if any, see below.

8 The declarator in a function definition specifies the name of the function being defined and the ~~identifiers and types~~ types (and optionally the names) of all the parameters; the declarator (possibly adjusted by an inferred type specifier) also serves as a function prototype for later calls to the same function in the same translation unit. The type of each parameter is adjusted as described in 6.7.8.3; the resulting type shall be a complete object type.

9 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

10 Each parameter has automatic storage duration; its identifier, if any,[254] is an lvalue.~~The layout of the storage for parameters is unspecified.~~[255]

11 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

12 After all parameters have been assigned, the compound statement of the function body is executed.

13 Unless otherwise specified, if the } that terminates the function body is reached, and the value of the function call is used by the caller, the behavior is undefined.

[253] For such a function, the scope of the function name only starts with the end of the first `return` statement, see 6.2.1. This means that such a function cannot be used for direct recursion before or within the first return statement.

[254] A parameter that has no declared name is inaccessible within the function body.

[255] A parameter identifier cannot be redeclared in the function body except in an enclosed block. As any object with automatic storage duration, each parameter gives rise to a unique storage instance representing it. Thus the relative layout of parameters in the address space is unspecified.

14 Provided the constraints above are respected, the return type of an underspecified function definition is adjusted as if *type* had been inserted in the definition. The type of the defined function is incomplete within the function body until a **`return`** statement is met.

15 **NOTE** In a function definition, the type of the function and its prototype cannot be inherited from a typedef:

```
typedef int F(void);          // type F is "function with no parameters
                              // returning int"
F f, g;                       // f and g both have type compatible with F
F f { /* ... */ }             // WRONG: syntax/constraint error
F g() { /* ... */ }           // WRONG: declares that g returns a function
int f(void) { /* ... */ }     // RIGHT: f has type compatible with F
int g() { /* ... */ }         // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }      // e returns a pointer to a function
F *((e))(void) { /* ... */ }  // same:  parentheses irrelevant
F f { /* ... */ }             // WRONG: syntax/constraint error
F g() { /* ... */ }           // WRONG: declares that g returns a function
int f(void) { /* ... */ }     // RIGHT: f has type compatible with F
int g() { /* ... */ }         // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }      // e returns a pointer to a function
F *((e))(void) { /* ... */ }  // same:  parentheses irrelevant
int (*fp)(void);              // fp points to a function that has type F
F *Fp;                        // Fp points to a function that has type F
```

16 **EXAMPLE 1** In the following:

```
extern int max(int a, int b)
{
        return a  >  b ? a: b;
}
```

`extern` is the storage-class specifier and **`int`** is the type specifier; `max(int a, int b)` is the function declarator; and

```
{ return a  >  b ? a: b; }
```

is the function body.

17 **EXAMPLE 2** To pass one function to another, one might say

```
        int f(void);
        /* ... */
        g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
        /* ... */
        (*funcp)(); /* or funcp(); ...*/
}
```

or, equivalently,

```
void g(int func(void))
{
        /* ... */
        func(); /* or (*func)(); ...*/
}
```

18 **EXAMPLE 3** Consider the following function that computes the maximum value of two parameters that have integer types T and S.

```
    inline auto max(T a, S b){
      return (a < 0)
```

```
             ? ((b < 0) ? ((a < b) ? b : a) : b)
             : ((b ≥ 0) ? ((a < b) ? b : a) : a);
    }
    ...
    extern auto max(T, S); // External declaration forces symbol emission
    auto max(T, S);        // same
    auto max();            // same
```

The **return** expression performs default arithmetic conversion to determine a type that can hold the maximum value and is at least as wide as **int**. The function definition is adjusted to that return type. This property holds regardless if types T and S have the same or different signedness.

The **extern** declaration and the equivalent ones are valid, because they follow the definition and thus the inferred return type is known.

19 **EXAMPLE 4** The following function computes the sum over an array of integers of type T and returns the value as the promoted type of T.

```
    auto sum(size_t n, T A[n]){
      switch(n) {
        case 0:
           return +((T)0);                              // return the promoted type
        case 1:
           return +A[0];                                // return the promoted type
        default:
           return sum(n/2, A) + sum(n - n/2, &A[n/2]);  // valid recursion
      }
    }
```

If instead sum would have bee defined with a prototype as follows

```
    T sum(size_t n, T A[n]);
```

for a narrow type T such as **unsigned char**, the return type and result would be different from the previous. In particular, the result of the addition would have been converted back from the promoted type to T before each **return**, possibly leading to a surprising overall results.

6.9.2 External object definitions

Semantics

1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to ~~{ 0 }~~ { }.

3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

4 **EXAMPLE 1**

```
int i1 = 1;         // definition, external linkage
static int i2 = 2;  // definition, internal linkage
extern int i3 = 3;  // definition, external linkage
int i4;             // tentative definition, external linkage
static int i5;      // tentative definition, internal linkage

int i1;             // valid tentative definition, refers to previous
int i2;             // 6.2.2 renders undefined, linkage disagreement
int i3;             // valid tentative definition, refers to previous
int i4;             // valid tentative definition, refers to previous
int i5;             // 6.2.2 renders undefined, linkage disagreement

extern int i1;      // refers to previous, whose linkage is external
extern int i2;      // refers to previous, whose linkage is internal
extern int i3;      // refers to previous, whose linkage is external
extern int i4;      // refers to previous, whose linkage is external
extern int i5;      // refers to previous, whose linkage is internal
```

5 **EXAMPLE 2** If at the end of the translation unit containing

```
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

6.10 Preprocessing directives

Syntax

1 *preprocessing-file:*
group$_{\text{opt}}$

group:
group-part
group group-part

group-part:
if-section
control-line
text-line
*non-directive*

if-section:
if-group elif-groups$_{\text{opt}}$ *else-group*$_{\text{opt}}$ *endif-line*

if-group:
 # if ~~*constant-expression*~~ *controlling-expression* *new-line group*$_{\text{opt}}$
 # ifdef *identifier new-line group*$_{\text{opt}}$
 # ifndef *identifier new-line group*$_{\text{opt}}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
 # elif ~~*constant-expression*~~ *controlling-expression* *new-line group*$_{\text{opt}}$

else-group:
 # else *new-line group*$_{\text{opt}}$

endif-line:
 # endif *new-line*

control-line:
 # include *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list*$_{\text{opt}}$ **)**
replacement-list new-line
 # define *identifier lparen* ~~**...**~~ **...** **)** *replacement-list new-line*
 # define *identifier lparen identifier-list* **,** ~~**...**~~ **...** **)**
replacement-list new-line
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens*$_{\text{opt}}$ *new-line*
 # pragma *pp-tokens*$_{\text{opt}}$ *new-line*
*new-line*

text-line:
pp-tokens$_{\text{opt}}$ *new-line*

non-directive:
pp-tokens new-line

lparen:
a **(** character not immediately preceded by white space

replacement-list:
pp-tokens$_{\text{opt}}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

identifier-list:
identifier
identifier-list **,** *identifier*

Description

2 A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a **#** preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence.[256) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

3 A text line shall not begin with a **#** preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

4 When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

5 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing **#** preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

Semantics

6 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

7 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

8 **EXAMPLE** In:

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a **#** at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced.

9 The execution of a non-directive preprocessing directive results in undefined behavior.

6.10.1 Conditional inclusion

Constraints

1 The ~~expression that controls~~ controlling expression of a conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below;[257) and it may contain unary operator expressions of the form

`defined` *identifier*

or

`defined (` *identifier* **`)`**

which evaluate to ~~1~~ **`true`** if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **`#define`** preprocessing directive without an intervening **`#undef`** directive with the same subject identifier), ~~0~~ **`false`** if it is not.

2 Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a

[256) Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the **#** character string literal creation operator in 6.10.3.2, for example).

[257) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

token (6.4).

Semantics

3 Preprocessing directives of the forms

> **# if** ~~*constant-expression*~~ *controlling-expression* *new-line* $group_{opt}$
> **# elif** ~~*constant-expression*~~ *controlling-expression* *new-line* $group_{opt}$

check whether the controlling constant expression evaluates to nonzero.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text. If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers other than **false** and **true** (including those lexically identical to keywords) are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax_t** and **uintmax_t** defined in the header `<stdint.h>`.[258] This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.[259] Also, whether a single-character character constant may have a negative value is implementation-defined.

5 Preprocessing directives of the forms

> **# ifdef** *identifier* *new-line* $group_{opt}$
> **# ifndef** *identifier* *new-line* $group_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and ~~**#if !defined**~~ **#if ¬defined** *identifier* respectively.

6 Each directive's condition is checked in order. If it evaluates to ~~false (zero)~~**false** , the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to ~~true (nonzero)~~ **true** is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to ~~true~~**true** , and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.[260]

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.20.1.5).

[258]Thus, on an implementation where **INT_MAX** is `0x7FFF` and **UINT_MAX** is `0xFFFF`, the constant `0x8000` is signed and positive within a **#if** expression even though it would be unsigned in translation phase 7.

[259]Thus, the controlling expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' ≡ 25
if ('z' - 'a' ≡ 25)
```

[260]As indicated by the syntax, no preprocessing tokens are allowed to follow a **#else** or **#endif** directive before the terminating new-line character. However, comments can appear anywhere in a source file, including within a preprocessing directive.

6.10.2 Source file inclusion

Constraints

1 A **#include** directive shall identify a header or source file that can be processed by the implementation.

Semantics

2 A preprocessing directive of the form

 # include < *h-char-sequence* **>** *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

 # include " *q-char-sequence* **"** *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

 # include < *h-char-sequence* **>** *new-line*

with the identical contained sequence (including > characters, if any) from the original directive.

4 A preprocessing directive of the form

 # include *pp-tokens* *new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.[261] The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (6.4.2.1) followed by a period (`.`) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

6 A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

7 **EXAMPLE 1** The most common uses of **#include** preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

8 **EXAMPLE 2** This illustrates macro-replaced **#include** directives:

```
#if VERSION == 1
#if VERSION ≡ 1
        #define INCFILE  "vers1.h"
#elif VERSION == 2
#elif VERSION ≡ 2
        #define INCFILE  "vers2.h"   // and so on
```

[261]Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

```
    #else
          #define INCFILE  "versN.h"
    #endif
    #include INCFILE
```

Forward references: macro replacement (6.10.3).

6.10.3 Macro replacement

Constraints

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an object-like macro shall not be redefined by another **`#define`** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **`#define`** preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

3 There shall be white space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the ~~`...`~~...). There shall exist a `)` preprocessing token that terminates the invocation.

5 The identifier **`__VA_ARGS__`** shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics

7 The identifier immediately following the **`define`** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

8 If a **#** preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

`# define` *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[262] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

`# define` *identifier lparen identifier-list*$_{opt}$ **`)`** *replacement-list new-line*
`# define` *identifier lparen* ~~`...`~~... **`)`** *replacement-list new-line*
`# define` *identifier lparen identifier-list* **`,`** ~~`...`~~... **`)`** *replacement-list new-line*

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **`#define`** preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing

[262]Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,[263)] the behavior is undefined.

12 If there is a ~~`...`~~... in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the ~~`...`~~...).

6.10.3.1 Argument substitution

1 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a **#** or **##** preprocessing token or followed by a **##** preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

2 An identifier **`__VA_ARGS__`** that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

6.10.3.2 The # operator

Constraints

1 Each **#** preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

2 If, in the replacement list, a parameter is immediately preceded by a **#** preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a `\` character is inserted before each `"` and `\` character of a character constant or string literal (including the delimiting `"` characters), except that it is implementation-defined whether a `\` character is inserted before the `\` character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is `""`. The order of evaluation of **#** and **##** operators is unspecified.

6.10.3.3 The ## operator

Constraints

1 A **##** preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a **##** preprocessing token, the parameter is replaced by the corresponding argument's preprocess-

263)Despite the name, a non-directive is a preprocessing directive.

ing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[264)]

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a **##** preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of **##** operators is unspecified.

4 **EXAMPLE** In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the **##** operator.

6.10.3.4 Rescanning and further replacement

1 After all parameters in the replacement list have been substituted and **#** and **##** processing has taken place, all placemarker preprocessing tokens are removed. The resulting preprocessing token sequence is then rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

4 **EXAMPLE** There are cases where it is not clear whether a replacement is nested or not. For example, given the following macro definitions:

```
#define f(a) a*g
#define g(a) f(a)
```

264) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

the invocation

```
f(2)(9)
```

could expand to either

```
2*f(9)
```

or

```
2*9*g
```

Strictly conforming programs are not permitted to depend on such unspecified behavior.

6.10.3.5 Scope of macro definitions

1 A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

2 A preprocessing directive of the form

 # undef *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 **EXAMPLE 1** The simplest use of this facility is to define a "manifest constant", as in

```
#define TABSIZE 100

int table[TABSIZE];
```

4 **EXAMPLE 2** The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a)  >  (b) ? (a): (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5 **EXAMPLE 3** To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)   f(x * (a))
#define f(a)   f(x × (a))
#undef  x
#define x      2
#define g      f
#define z      z[0]
#define h      g(\~{ }
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
      (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
        f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
        f(2 * (2+(3,4)-0,1)) | f(2 * (\~{ } 5)) & f(2 * (0,1))^m(0,1);
        f(2 × (y+1)) + f(2 × (f(2 × (z[0])))) % f(2 × (0)) + t(1);
        f(2 × (2+(3,4)-0,1)) | f(2 × (\~{ } 5)) & f(2 × (0,1))^m(0,1);
        int i[] = { 1, 23, 4, 5,  };
        char c[2][6] = { "hello", "" };
```

6 **EXAMPLE 4** To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                        x ## s, x ## t)
#define INCFILE(n)  vers ## n
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
            == 0) str(: @\n), s);
            ≡ 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
  "strncmp(\"abc\\0d\", \"abc\", '\\4') ≡ 0" ": @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
  "strncmp(\"abc\\0d\", \"abc\", '\\4') ≡ 0: @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the **#** and **##** tokens in the macro definition is optional.

7 **EXAMPLE 5** To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
            10, 11, 12,  };
```

8 **EXAMPLE 6** To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)   (a)
#define FUNC_LIKE(a)(   /* note the white space */ \
                    a /* other stuff on this line
                       */)
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)     // different token sequence
#define OBJ_LIKE    (1 - 1) // different white space
#define FUNC_LIKE(b) (a)    // different parameter usage
#define FUNC_LIKE(b) (b)    // different parameter spelling
```

9 **EXAMPLE 7** Finally, to show the variable argument list macro facilities:

```
#define debug(...)       fprintf(stderr, __VA_ARGS__)
#define showlist(...)    puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...)    puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                    printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr,  "Flag");
fprintf(stderr,  "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):
            printf("x is %d but y is %d", x, y));
```

6.10.4 Line control

Constraints

1 The string literal of a **#line** directive, if present, shall be a character string literal.

Semantics

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token.

3 If a preprocessing token (in particular **__LINE__**) spans two or more physical lines, it is unspecified which of those line numbers is associated with that token. If a preprocessing directive spans two or more physical lines, it is unspecified which of those line numbers is associated with the preprocessing directive. If a macro invocation spans multiple physical or logical lines, it is unspecified which of those line numbers is associated with that invocation. The line number of a preprocessing token is independent of the context (in particular, as a macro argument or in a preprocessing directive). The line number of a **__LINE__** in a macro body is the line number of the macro invocation.

4 A preprocessing directive of the form

 # line *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647.

5 A preprocessing directive of the form

 # line *digit-sequence* **"** *s-char-sequence*$_{\text{opt}}$ **"** new-line

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

6 A preprocessing directive of the form

 # line *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **`line`** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.[265]

Recommended practice

7 The line number associated with a *pp-token* should be the line number of the first character of the *pp-token*. The line number associated with a preprocessing directive should be the line number of the line with the first **#** token. The line number associated with a macro invocation should be the line number of the first character of the macro name in the invocation.

6.10.5 Error directive

Semantics

1 A preprocessing directive of the form

> **`# error`** *pp-tokens*$_{\text{opt}}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

6.10.6 Pragma directive

Semantics

1 A preprocessing directive of the form

> **`# pragma`** *pp-tokens*$_{\text{opt}}$ *new-line*

where the preprocessing token **`STDC`** does not immediately follow **`pragma`** in the directive (prior to any macro replacement)[266] causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **`pragma`** that is not recognized by the implementation is ignored.

2 If the preprocessing token **`STDC`** does immediately follow **`pragma`** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[267] whose meanings are described elsewhere:

> **`# pragma`** **`STDC`** **`FP_CONTRACT`** *on-off-switch*
> **`# pragma`** **`STDC`** **`FENV_ACCESS`** *on-off-switch*
> **`# pragma`** **`STDC`** **`CX_LIMITED_RANGE`** *on-off-switch*
> **`# pragma`** **`CORE`** **`FUNCTION_ATTRIBUTE`** *attribute*
> **`# pragma`** **`CORE`** **`FUNCTION_ATTRIBUTE`** *identifier* **`OFF`**

on-off-switch: one of
> **`ON`** **`OFF`** **`DEFAULT`**

Forward references: the **`FP_CONTRACT`** pragma (7.12.2), the **`FENV_ACCESS`** pragma (7.6.1), the **`CX_LIMITED_RANGE`** pragma (7.3.4).

6.10.7 Null directive

Semantics

1 A preprocessing directive of the form

---

265)Because a new-line is explicitly included as part of the **`#line`** directive, the number of new-line characters read while processing to the first *pp-token* can be different depending on whether or not the implementation uses a one-pass preprocessor. Therefore, there are two possible values for the line number following a directive of the form **`#line __LINE__`** *new-line*.

266)An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **`STDC`** immediately follows **`pragma`**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

267)See "future language directions" (6.11.7).

*new-line*

has no effect.

6.10.8 Predefined macro names

1 The values of the predefined feature macros listed in the following subclauses[268] (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.

2 None of ~~these~~ the macro names, nor the identifier **`defined`**, shall be the subject of a **`#define`** or a **`#undef`** preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

3 The implementation shall not predefine the macro `__cplusplus`, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory feature macros

1 The following macro names shall be defined by the implementation:

`__DATE__` The date of translation of the preprocessing translation unit: a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the **`asctime`** function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

`__FILE__` The presumed name of the current source file (a character string literal).[269]

`__LINE__` The presumed line number (within the current source file) of the current source line (an integer constant).[269]

~~`__STDC__`~~ `__CORE__` The integer constant `1`, intended to indicate a conforming implementation of the core.

`__STDC_HOSTED__` The integer constant `1` if the implementation is a hosted implementation or the integer constant `0` if it is not.

~~`__STDC_VERSION__`~~ `__CORE_ALIAS_OVERWRITES__` **`true`** if using the **`core::alias`** attribute on an identifier with external linkage inhibits an external definition in any other translation units, **`false`** otherwise.

`__CORE_VERSION__` The integer constant *202005*`L`.[270]

`__TIME__` The time of translation of the preprocessing translation unit: a character string literal of the form `"hh:mm:ss"` as in the time generated by the **`asctime`** functions. If the time of translation is not available, an implementation-defined valid time shall be supplied. ~~the **`asctime`** functions (7.27.3.1).~~

~~The following macro names are conditionally defined by the implementation:~~

`__STDC_ISO_10646__` An integer constant of the form `yyyymmL` (for example, `199712L`). ~~If this symbol is defined, then every~~ Every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. ~~If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.~~

[268] See "future language directions" (6.11.8).
[269] The presumed source file name and line number can be changed by the **`#line`** directive.
[270] See Annex M for the values of analogous macro `__STDC_VERSION__` in previous revisions of the C standard. The intention is that this will remain an integer constant of type **`long int`** that is increased with each revision of this document.

~~__STDC_MB_MIGHT_NEQ_WC__ The integer constant 1, intended to indicate that, in the encoding for , a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.~~

__STDC_UTF_16__ The integer constant `1`, intended to indicate that values of type **char16_t** are UTF–16 encoded. ~~If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.~~

__STDC_UTF_32__ The integer constant `1`, intended to indicate that values of type **char32_t** are UTF–32 encoded. ~~If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.~~

2 Additionally there shall be macros **__CORE_*X*_IS_TYPE__** for each of the integer types defined in 6.2.5.1, where *X* is replaced by the all caps name of the type without the **_t** suffix, such that the macro expands to **true** if the type is a proper type, and **false** otherwise. For example, there is a macro **__CORE_CHAR8_IS_TYPE__** if the type **char8_t** is provided as a type that is different from **char**, **signed char**, or **unsigned char**.

3 **NOTE** In the C/C++ core the macros **__STDC_UTF_16__** and **__STDC_UTF_32__** are useless features since here it is assumed that the corresponding types are UTF encoded, anyhow. C also has an optional macro **__STDC_MB_MIGHT_NEQ_WC__** that would be set if the encodings of **wchar_t** and `code` would not agree on the basic character set.

Forward references: ~~common definitions (7.19~~the **asctime** functions (7.27.3.1), unicode utilities (7.28).

6.10.8.2 Conditional feature macros

1 The following macro names are conditionally defined by the implementation:

__STDC_ANALYZABLE__ The integer constant `1`, intended to indicate conformance to the specifications in Annex L (Analyzability).

__STDC_IEC_559__ The integer constant `1`, intended to indicate conformance to the specifications in Annex F (IEC 60559 floating-point arithmetic).

~~__STDC_IEC_559_COMPLEX__ The integer constant 1, intended to indicate adherence to the specifications in ?? (IEC 60559 compatible complex arithmetic).~~

~~__STDC_LIB_EXT1__ The integer constant *202005*L, intended to indicate support for the extensions defined in ?? (Bounds-checking interfaces).~~

~~__STDC_NO_ATOMICS__~~ **__CORE_NO_ATOMICS__** The integer constant `1`, intended to indicate that the implementation does not support ~~atomic types (including the _Atomic type qualifier) and~~ the `<stdatomic.h>` header.

~~__STDC_NO_COMPLEX__~~ **__CORE_NO_COMPLEX__** The integer constant `1`, intended to indicate that the implementation does not support ~~complex types or~~ the `<complex.h>` header.

__STDC_NO_THREADS__ The integer constant `1`, intended to indicate that the implementation does not support the `<threads.h>` header.

~~__STDC_NO_VLA__~~ **__CORE_NO_VLA__** The integer constant `1`, intended to indicate that the implementation does not support ~~variable length arrays or variably modified types.~~ definitions of variable length array type in block scope.

__CORE_WEAK_THREEWAY_COMPARISON__ An integer constant expression that indicates that some of the implementation-defined choices concerning three-way comparison of pointers, integers and structure types, leads to weak three-way comparison types. The possible values are described in 7.31.1.

2 **NOTE** For C, the absence of complex types is condionned by the feature test macro **__STDC_NO_COMPLEX__**, for C++ they are conditioned to the inclusion of a specific header. For this core specification we opted for simplicity of usage, so all operational language features for complex types are supposed to be available by default.

6.10.8.3 Mandatory type and value macros

1 ~~An implementationthat defines `__STDC_NO_COMPLEX__` shall not define `__STDC_IEC_559_COMPLEX__`.~~ The following macro names provide access to principal language features and shall be defined by the implementation.

`NULL` expands to an implementation-defined null pointer constant.

`tonullptr``(`*expr*`)` where the expression *expr* shall have a scalar type. If `expr` is a null pointer constant, the result is **`nullptr`**. Otherwise, the result is *expr*.[271]

`offsetof``(`*type*, *member-designator*`)` expands to an integer constant expression that has type **`size_t`**, the value of which is the offset in bytes, to the subobject (designated by *member-designator*), from the beginning of any object of type *type*. The type and member designator shall be such that given

```
static type t;
```

then the expression `&(t.` *member-designator*`)` evaluates to an address constant. If the specified *type* defines a new type, the behavior is undefined.

`atomic_type``(`*type-name*`)` specifies the atomic type that is derived from the type name, see 6.7.2.4.

`complex_type``(`*type-or-expression*`)` specifies for any arithmetic type or expression the complex type of least precision to which it is converted in usual arithmetic. It is equivalent to the type specification

```
decltype(((type-or-expression)+0) + 0.0if)
```

`generic_type``(`*expression*`)` specifies the generic type of *expression*. It is equivalent to the type specification

```
decltype([](auto x){ return x; }(expression))
```

`generic_value``(`*expression*`)` computes the value of a scalar expression, an array or a function designator. It is equivalent to

```
((generic_type(expression))(expression))
```

and is a constant expression, whenever *expression* is.

`issame``(`*expr0*, *expr1*`)` Returns **`true`** if the generic types of *expr0* and `expr1` are compatible, **`false`** otherwise.

`floating_value``(`*expression*`)` converts an arithmetic expression to the floating point type with least precision as by usual arithmetic conversion. It is equivalent to

```
((expression)+0.0f)
```

and is a constant expression, whenever *expression* is.

`complex_value``(`*expression*`)` computes the complex value of an arithmetic expression. It is equivalent to

[271] The intent of this macro is to normalize values that might be passed to interfaces that expect pointers. All null pointer constants are converted to **`nullptr`** such that they can be captured by a **`nullptr_t`** case of a **`generic_selection`**.

```
((expression)+0.0if)
```

and is a constant expression, whenever *expression* is.

`real_type``(`*type-or-expression*`)` specifies the real type of an arithmetic expression or type name.

`real_value``(`*expression*`)` computes the real value of an arithmetic expression. It has floating point type and is equivalent to

```
((real_type(complex_value(expression)))expression)
```

and is a constant expression, whenever *expression* is.

`imaginary_value``(`*expression*`)` computes the imaginary value of an arithmetic expression. It has floating point type and is equivalent to

```
((real_type(complex_value(expression)))(-1.0if × (expression)))
```

and is a constant expression, whenever *expression* is.

6.10.8.4 Optional keyword macros

1 The keywords

`alignas`	**`bool`**	**`not`**	**`true`**
`alignof`	**`compl`**	**`nullptr`**	**`xor_eq`**
`and_eq`	**`decltype`**	**`or_eq`**	**`xor`**
`and`	**`false`**	**`or`**	
`bitand`	**`generic_selection`**	**`static_assert`**	
`bitor`	**`not_eq`**	**`thread_local`**	

optionally are also predefined macro names that expand to unspecified tokens.

6.10.9 Pragma operator

Semantics

1 A unary operator expression of the form:

`_Pragma` **`(`** *string-literal* **`)`**

is processed as follows: The string literal is *destringized* by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 **EXAMPLE** A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\\listing.dir\"")
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING (..\listing.dir)
```

6.11 Future language directions

6.11.1 Floating types

1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **`long double`**.

6.11.2 Linkages of identifiers

1 Declaring an identifier with internal linkage at file scope without the **`static`** storage-class specifier is an obsolescent feature.

6.11.3 External names

1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Storage-class specifiers

1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.6 Function declarators

1 The use of function declarators without prototypes is an obsolescent feature.

6.11.7 Pragma directives

1 Pragmas whose first preprocessing token is **`STDC`** are reserved for future standardization.

6.11.8 Predefined macro names

1 Macro names beginning with **`__STDC_`** and **`__CORE_`** are reserved for future standardization.

7. Library

7.1 Introduction

7.1.1 Definitions of terms

1 A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The *length of a string* is the number of ~~bytes~~ characters preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

2 The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[272] It is represented in the text and examples by a period, but may be changed by the `setlocale` function.

3 A *null wide character* is a wide character with code value zero.

4 A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

5 A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state (see 5.2.1.1). A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.[273] In this clause, references to "*white-space character*" refer to (execution) white-space character as defined by `isspace`. References to "*white-space wide character*" refer to (execution) white-space wide character as defined by `iswspace`.

Forward references: character handling (7.4), the `setlocale` function (7.11.1.1).

7.1.2 Standard headers

1 Each library function is declared, with a type that includes a prototype, in a *header*,[274] whose contents are made available by the `#include` preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use. ~~In addition to the provisions given in this clause, an implementation that defines `__STDC_LIB_EXT1__` shall conform to the specifications in ?? and Subclause ?? should be read as if it were merged into the parallel structure of named subclauses of this clause. Declarations of types described here or in ?? shall not include type qualifiers, unless explicitly stated otherwise.~~

2 The standard headers are[275]

`<assert.h>`	`<limits.h>`	`<stdint.h>`
`<complex.h>`	`<locale.h>`	`<stdio.h>`
`<ctype.h>`	`<math.h>`	`<stdlib.h>`
`<errno.h>`	`<setjmp.h>`	`<stdnoreturn.h>`
`<fenv.h>`	`<signal.h>`	`<string.h>`
`<float.h>`	`<stdarg.h>`	`<threads.h>`
`<inttypes.h>`	`<stdatomic.h>`	`<time.h>`

[272] The functions that make use of the decimal-point character are the numeric conversion functions (7.22.1) and the formatted input/output functions (7.21.6, 7.29.2).

[273] For state-dependent encodings, the values for `MB_CUR_MAX` and `MB_LEN_MAX` are thus required to be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation's choice.

[274] A header is not necessarily a source file, nor are the < and > delimited sequences in header names necessarily valid source file names.

[275] The headers `<complex.h>`, `<stdatomic.h>`, and `<threads.h>` are conditional features that implementations need not support; see 6.10.8.2.

`<uchar.h>` `<wchar.h>` `<wctype.h>`

3 Additionally, the empty headers `<iso646.h>`, `<stdalign.h>`, `<stdbool.h>`, `<stddef.h>`, and `<tgmath.h>` shall be present for backwards compatibility reasons only.

4 If a file with the same name as one of the above < and > delimited sequences, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files, the behavior is undefined.

5 Standard headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including `<assert.h>` depends on the definition of **NDEBUG** (see 7.2). If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion of the header or when any macro defined in the header is expanded.

6 Some standard headers define or declare identifiers that had not been present in previous versions of this document. To allow implementations and users to adapt to that situation, they also define a version macro for feature test of the form **__STDC_VERSION_*XXXX*_H__** which expands to *202005*L, where XXXX is the all-caps spelling of the corresponding header `<xxxx.h>`.

7 Any definition of an object-like macro described in this clause ~~or ??~~ shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

8 Any declaration of a library function shall have external linkage.

9 A summary of the contents of the standard headers is given in Annex B.

Forward references: diagnostics (7.2).

7.1.3 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

— All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use, except those identifiers which are lexically identical to keywords.[276)]

— All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

— Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 7.1.4).

— All identifiers with external linkage in any of the following subclauses (including the future library directions) and **errno** are always reserved for use as identifiers with external linkage.[277)]

— Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

276) Allows identifiers spelled with a leading underscore followed by an uppercase letter that match the spelling of a keyword to be used as macro names by the program.

277) The list of reserved identifiers with external linkage includes **math_errhandling**, **setjmp**, **va_copy**, and **va_end**.

2 No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier or attribute token described in 6.7.15 as a macro name, the behavior is undefined.

3 If the program removes (with **#undef**) any macro definition of an identifier in the first group listed above or attribute token described in 6.7.15, the behavior is undefined.

7.1.4 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow:

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to a non-modifiable storage instance when the corresponding parameter is not const-qualified) or a type (after default argument promotion) not expected by a function with a variable number of arguments, the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

— Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[278] The use of **#undef** to remove any macro definition will also ensure that an actual function is referred to.

— Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.[279]

— Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.[280]

— All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in **#if** preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

[278] This means that an implementation is required to provide an actual function for each library function, even if it also provides a macro for that function.

[279] Such macros might not contain the sequence points that the corresponding function calls do.

[280] Because external identifiers and some macro names beginning with an underscore are reserved, implementations can provide special semantics for such names. For example, the identifier `_BUILTIN_abs` could be used to indicate generation of in-line code for the **abs** function. Thus, the appropriate header could specify

```
#define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as **abs** will be a genuine function can write

```
#undef abs
```

whether the implementation's header provides a macro implementation of **abs** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.[281)]

5 Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments.[282)] Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

6 Unless otherwise specified, library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.[283)]

7 Unless otherwise specified, library functions by themselves do not expose storage instances, but library functions that execute application specific callbacks[284)] may expose storage instances through calls into these callbacks.

8 **EXAMPLE** The function **atoi** can be used in any of several ways:

— by use of its associated header (possibly generating a macro expansion)

```
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

— by use of its associated header (assuredly generating a true function reference)

```
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

— by explicit declaration

```
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

[281)]Thus, a signal handler cannot, in general, call standard library functions.

[282)]This means, for example, that an implementation is not permitted to use a **static** object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. Similarly, an implementation of **memcpy** is not permitted to copy bytes beyond the specified length of the destination object and then restore the original values because it could cause a data race if the program shared those bytes between threads.

[283)]This allows implementations to parallelize operations if there are no visible side effects.

[284)]The following library functions call application specific functions that they or related functions receive as arguments: **bsearch**, **call_once**, **exit** (for **atexit** handlers), **qsort**, **quick_exit** (for **at_quick_exit** handlers), and **thrd_exit** (for thread specific storage).

7.2 Diagnostics `<assert.h>`

1 The header `<assert.h>` defines the **`assert`** ~~and **`static_assert`** macros~~ macro and refers to another macro,

```
NDEBUG
```

which is *not* defined by `<assert.h>`. If **`NDEBUG`** is defined as a macro name at the point in the source file where `<assert.h>` is included, the **`assert`** macro is defined simply as

```
#define assert(ignore) ((void)0)
```

The **`assert`** macro is redefined according to the current state of **`NDEBUG`** each time that `<assert.h>` is included.

2 The **`assert`** macro shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

~~The macro expands to **`_Static_assert`**.~~

7.2.1 Program diagnostics

7.2.1.1 The **`assert`** macro

Synopsis

1
```
#include <assert.h>
void assert(scalar expression);
void assert(scalar expression) [[ core::evaluates(stderr) ]] ;
```

Description

2 The **`assert`** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if `expression` (which shall have a scalar type) is false (that is, compares equal to 0), the **`assert`** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros **`__FILE__`** and **`__LINE__`** and of the identifier **`__func__`**) on the standard error stream in an implementation-defined format.[285] It then calls the **`abort`** function.

Returns

3 The **`assert`** macro returns no value.

Forward references: the **`abort`** function (7.22.5.1).

[285]The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

7.3 Complex arithmetic <complex.h>

7.3.1 Introduction

1 The header <complex.h> defines macros and ~~declares functions that~~ amends type-generic macros that are otherwise defined in the <math.h> to support complex arithmetic. The <math.h> is implicitly included. If both headers are included, the order in which they are included shall not impact on the syntax or sematics of the interfaces that are provided.

2 Implementations that define the macro ~~__STDC_NO_COMPLEX__~~ __CORE_NO_COMPLEX__ need not provide this header nor support any of its facilities.

3 ~~Each synopsis, other than for the **CMPLX** macros, specifies a family of functions~~ The subclauses below add functionality for complex arguments to type-generic macros were the principal definition is described in 7.12. The functionality is added analogously as for real floating arguments and conversions to functions with real floating arguments, there, with some specificities for the complex value case as stated. Obsolecent function names are also reserved, consisting of a principal function ~~with one or more **double complex**~~ (of the name prefixed with the character “c”) and with **complex_type(double)** parameters and a ~~**double complex** or **double**~~ **complex_type(double)** return value; and other functions with the same name but with **f** and **l** suffixes which are corresponding functions with **float** and **long double** parameters and return values. The identifiers that are such reserved are

cabsf	casinf	catanh	cexpf	csinf	ctanf
cabsl	casinhf	catanl	cexpl	csinhf	ctanhf
cabs	casinhl	catan	cexp	csinhl	ctanhl
cacosf	casinh	ccosf	clogf	csinh	ctanh
cacoshf	casinl	ccoshf	clogl	csinl	ctanl
cacoshl	casin	ccoshl	clog	csin	ctan
cacosh	catanf	ccosh	cpowf	csqrtf	
cacosl	catanhf	ccosl	cpowl	csqrtl	
cacos	catanhl	ccos	cpow	csqrt	

4 ~~The macro~~ The feature test macro **__CORE_VERSION_COMPLEX_H__** expands to the token *202005*L. Additionally reserved are also the obsolecent macros ~~expands to either **_Imaginary_I** or **_Complex_I**. If **_Imaginary_I** is not defined, I~~

complex	_Complex_I	I	CMPLX	CMPLXF	CMPLXL

~~shall expand to **_Complex_I**.~~

5 ~~Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros **complex**, **imaginary**, and I. the **CMPLX** macros (??), IEC 60559-compatible complex arithmetic (??).~~ Core function attributes are implied analogously to the previsions in 7.12.

7.3.2 Conventions

1 Values are interpreted as radians, not degrees. An implementation may set **errno** but is not required to.

7.3.3 Branch cuts

1 Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications ~~of ??,~~ the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part -0, maps to the negative imaginary axis.

2 Implementations that do not support a signed zero (see Annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is

approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

7.3.4 The `CX_LIMITED_RANGE` pragma

Synopsis

1
```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Description

2 The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The `CX_LIMITED_RANGE` pragma can be used to inform the implementation that (where the state is "on") the usual mathematical formulas are acceptable.[286] The pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is "off".

7.3.5 Trigonometric functions

1 ~~Synopsis replace~~

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

The `cos`, `sin`, and `tan` type-generic macros (7.12.4) are extended to complex arguments by implementing the appropiate definitions for the complex domain. For other functions, specific precautions apply according to the following clauses.

7.3.5.1 The `acos` type-generic macro

Description

1 ~~The~~ For complex arguments, the `acos` type-generic macro ~~compute~~ computes the complex arc cosine of `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

2 ~~The~~ For complex arguments, the `acos` type-generic macro ~~return~~ returns the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

7.3.5.2 The `asin` type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
```

[286]The purpose of the pragma is to allow the implementation to use the formulas:

$$
\begin{aligned}
(x+iy)\times(u+iv) &= (xu-yv)+i(yu+xv)\\
(x+iy)\,/\,(u+iv) &= [(xu+yv)+i(yu-xv)]/(u^2+v^2)\\
|x+iy| &= \sqrt{x^2+y^2}
\end{aligned}
$$

where the programmer can determine they are safe.

```
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **asin** type-generic macro ~~compute~~ computes the complex arc sine of z, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

2 ~~The~~ For complex arguments, the **asin** type-generic macro ~~return~~ returns the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ along the real axis.

7.3.5.3 The atan type-generic macro

~~Synopsis~~ replace

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **atan** type-generic macro ~~compute~~ computes the complex arc tangent of z, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

2 ~~The~~ For complex arguments, the **atan** type-generic macro ~~return~~ returns the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ along the real axis.

~~Synopsis replace~~

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

~~**Description**~~

~~The compute the complex cosine of z.~~

~~**Returns**~~

~~The return the complex cosine value.~~

~~Synopsis replace~~

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

~~**Description**~~

~~The compute the complex sine of z.~~

~~**Returns**~~

~~The return the complex sine value.~~

~~Synopsis replace~~

```
#include <complex.h>
double complex ctan(double complex z);
```

```
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

~~**Description**~~

~~The compute the complex tangent of z.~~

~~**Returns**~~

~~The return the complex tangent value.~~

7.3.6 Hyperbolic functions

1 The **cosh**, **sinh**, and **tanh** type-generic macros (7.12.5) are extended to complex arguments by implementing the appropiate definitions for the complex domain. For other functions, specific precautions apply according to the following clauses.

7.3.6.1 The **acosh** type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **acosh** type-generic macro ~~compute~~ computes the complex arc hyperbolic cosine of z, with a branch cut at values less than 1 along the real axis.

Returns

2 ~~The~~ For complex arguments, the **acosh** type-generic macro ~~return~~ returns the complex arc hyperbolic cosine value, in the range of a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.6.2 The **asinh** type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **asinh** type-generic macro ~~compute~~ computes the complex arc hyperbolic sine of z, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

2 ~~The~~ For complex arguments, the **asinh** type-generic macro ~~return~~ returns the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-\frac{i\pi}{2}, +\frac{i\pi}{2}]$ along the imaginary axis.

7.3.6.3 The **atanh** type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **atanh** type-generic macro ~~compute~~ computes the complex arc hyperbolic tangent of `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

2 ~~The~~ For complex arguments, the **atanh** type-generic macro ~~return~~ returns the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-\frac{i\pi}{2}, +\frac{i\pi}{2}]$ along the imaginary axis.

~~Synopsis replace~~

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

~~**Description**~~

~~The compute the complex hyperbolic cosine of `z`.~~

~~**Returns**~~

~~The return the complex hyperbolic cosine value.~~

~~Synopsis replace~~

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

~~**Description**~~

~~The compute the complex hyperbolic sine of `z`.~~

~~**Returns**~~

~~The return the complex hyperbolic sine value.~~

~~Synopsis replace~~

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

~~**Description**~~

~~The compute the complex hyperbolic tangent of `z`.~~

~~**Returns**~~

~~The return the complex hyperbolic tangent value.~~

7.3.7 Exponential and logarithmic functions

~~Synopsis replace~~

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

~~**Description**~~

1 ~~The compute the complex base-e exponential of `z`.~~

~~**Returns**~~

~~The return the complex base-e exponential value.~~ The **exp** type-generic macro (7.12.6.1) is extended to complex arguments by implementing the appropiate definition for the complex domain. For the **log** function specific precautions apply according to the following clause.

7.3.7.1 The `log` type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **log** type-generic macro ~~compute~~ computes the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

Returns

2 ~~The~~ For complex arguments, the **log** type-generic macro ~~return~~ returns the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.8 Power and absolute-value functions

~~Synopsis replace~~

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

~~**Description**~~

~~The compute the complex absolute value (also called norm, modulus, or magnitude) of `z`.~~

~~**Returns**~~

~~The return the complex absolute value.~~

7.3.8.1 The `pow` type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
```

Description

1 ~~The~~ For complex arguments, the **pow** type-generic macro ~~compute~~ computes the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

Returns

2 ~~The~~ For complex arguments, the **pow** type-generic macro ~~return~~ returns the complex power function value.

7.3.8.2 The `sqrt` type-generic macro

~~Synopsis replace~~

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Description

1 ~~The~~ For complex arguments, the **sqrt** type-generic macro ~~compute~~ computes the complex square root of `z`, with a branch cut along the negative real axis.

Returns

2 ~~The~~ For complex arguments, the **sqrt** type-generic macro ~~return~~ returns the complex square root value, in the range of the right half-plane (including the imaginary axis)~~.~~

~~Synopsis replace~~

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

~~**Description**~~

~~The compute the argument (also called phase angle) of `z`, with a branch cut along the negative real axis.~~

~~**Returns**~~

~~The return the value of the argument in the interval $[-\pi, +\pi]$.~~

~~Synopsis replace~~

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

~~**Description**~~

~~The compute the imaginary part of `z`.~~

~~**Returns**~~

~~The return the imaginary part value (as a real).~~

~~Synopsis replace~~

```
#include <complex.h>
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

~~**Description**~~

~~The expand to an expression of the specified complex type, with the real part having the (converted) value of `x` and the imaginary part having the (converted) value of `y`. The resulting expression shall be suitable for use as an initializer for an object with static or thread storage duration, provided both arguments are likewise suitable.~~

~~**Returns**~~

~~The return the complex value `x` + i`y`. These macros act as if the implementation supported imaginary types and the definitions were:~~

~~Synopsis replace~~

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

~~**Description**~~

~~The compute the complex conjugate of `z`, by reversing the sign of its imaginary part.~~

~~**Returns**~~

~~The return the complex conjugate value.~~

~~Synopsis replace~~

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

~~**Description**~~

~~The compute a projection of `z` onto the Riemann sphere: `z` projects to `z` except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If `z` has an infinite part, then **`cproj`**`(z)` is equivalent to~~

~~**Returns**~~

~~The return the value of the projection onto the Riemann sphere.~~

~~Synopsis replace~~

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

~~**Description**~~

~~The compute the real part of `z`.~~

~~**Returns**~~

~~The return the real part value~~.

7.4 Character handling <ctype.h>

1 The header <ctype.h> declares several functions useful for classifying and mapping characters.[287] In all cases the argument is an **int**, the value of which shall be representable as an **unsigned char** or shall equal the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

2 The behavior of most of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the "C" locale are noted below.

3 The term *printing character* refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.[288] All letters and digits are printing characters.

4 Attributes corresponding to the pragma

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced
```

is implied for the whole header. Additionally the pragma

```
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(locale)
```

is implied for all functions other than **isdigit** and **isxdigit**. For the latter two, if the argument has any other value than forseen above and if is a constant expression, the call is erroneous and a diagnostic shall be issued; if it has another value and is not a constant expression, the behavior is undefined.

Forward references: **EOF** (7.21.1), localization (7.11).

7.4.1 Character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument c conforms to that in the description of the function.

7.4.1.1 The isalnum function

Synopsis

1
```
#include <ctype.h>
int isalnum(int c);
```

Description

2 The **isalnum** function tests for any character for which **isalpha** or **isdigit** is true.

7.4.1.2 The isalpha function

Synopsis

1
```
#include <ctype.h>
int isalpha(int c);
```

Description

2 The **isalpha** function tests for any character for which **isupper** or **islower** is true, or any character that is one of a locale-specific set of alphabetic characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true.[289] In the "C" locale, **isalpha** returns true only for the characters for which **isupper** or **islower** is true.

[287]See "future library directions" (7.32.1).

[288]In an implementation that uses the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde); the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL).

[289]The functions **islower** and **isupper** test true or false separately for each of these additional characters; all four combinations are possible.

7.4.1.3 The `isblank` function

Synopsis

1
```
#include <ctype.h>
int isblank(int c);
```

Description

2 The **`isblank`** function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which **`isspace`** is true and that is used to separate words within a line of text. The standard blank characters are the following: space (`' '`), and horizontal tab (`'\t'`). In the `"C"` locale, **`isblank`** returns true only for the standard blank characters.

7.4.1.4 The `iscntrl` function

Synopsis

1
```
#include <ctype.h>
int iscntrl(int c);
```

Description

2 The **`iscntrl`** function tests for any control character.

7.4.1.5 The `isdigit` function

Synopsis

1
```
#include <ctype.h>
int isdigit(int c);
constexpr int isdigit(int c);
```

Description

2 The **`isdigit`** function tests for any decimal-digit character (as defined in 5.2.1).

7.4.1.6 The `isgraph` function

Synopsis

1
```
#include <ctype.h>
int isgraph(int c);
```

Description

2 The **`isgraph`** function tests for any printing character except space (`' '`).

7.4.1.7 The `islower` function

Synopsis

1
```
#include <ctype.h>
int islower(int c);
```

Description

2 The **`islower`** function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of **`iscntrl`**, **`isdigit`**, **`ispunct`**, or **`isspace`** is true. In the `"C"` locale, **`islower`** returns true only for the lowercase letters (as defined in 5.2.1).

7.4.1.8 The `isprint` function

Synopsis

1
```
#include <ctype.h>
int isprint(int c);
```

Description

2 The `isprint` function tests for any printing character including space (`' '`).

7.4.1.9 The `ispunct` function

Synopsis

1
```
#include <ctype.h>
int ispunct(int c);
```

Description

2 The `ispunct` function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither `isspace` nor `isalnum` is true. In the `"C"` locale, `ispunct` returns true for every printing character for which neither `isspace` nor `isalnum` is true.

7.4.1.10 The `isspace` function

Synopsis

1
```
#include <ctype.h>
int isspace(int c);
```

Description

2 The `isspace` function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which `isalnum` is false. The standard white-space characters are the following: space (`' '`), form feed (`'\f'`), new-line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). In the `"C"` locale, `isspace` returns true only for the standard white-space characters.

7.4.1.11 The `isupper` function

Synopsis

1
```
#include <ctype.h>
int isupper(int c);
```

Description

2 The `isupper` function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the `"C"` locale, `isupper` returns true only for the uppercase letters (as defined in 5.2.1).

7.4.1.12 The `isxdigit` function

Synopsis

1
```
#include <ctype.h>
int isxdigit(int c);
constexpr int isxdigit(int c);
```

Description

2 The `isxdigit` function tests for any hexadecimal-digit character (as defined in 6.4.4.1).

7.4.2 Character case mapping functions

7.4.2.1 The `tolower` function

Synopsis

1
```
#include <ctype.h>
int tolower(int c);
```

Description

2 The `tolower` function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a character for which **`isupper`** is true and there are one or more corresponding characters, as specified by the current locale, for which **`islower`** is true, the **`tolower`** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.4.2.2 The `toupper` function

Synopsis

1
```
#include <ctype.h>
int toupper(int c);
```

Description

2 The **`toupper`** function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a character for which **`islower`** is true and there are one or more corresponding characters, as specified by the current locale, for which **`isupper`** is true, the **`toupper`** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.5 Errors <errno.h>

1 The header <errno.h> defines several macros, all relating to the reporting of error conditions.

2 The macros are

```
EDOM
EILSEQ
ERANGE
```

which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in **#if** preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue[290] that has type **int** and thread local storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **errno**, the behavior is undefined.

3 The value of **errno** in the initial thread is zero at program startup (the initial value of **errno** in other threads is an indeterminate value), but is never set to zero by any library function.[291] The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in this document.

4 Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter,[292] may also be specified by the implementation.

290)The macro **errno** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, ∗**errno**()).

291)Thus, a program that uses **errno** for error checking would set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **errno** on entry and then set it to zero, as long as the original value is restored if **errno**'s value is still zero just before the return.

292)See "future library directions" (7.32.2).

7.6 Floating-point environment `<fenv.h>`

1 The header `<fenv.h>` defines several macros, and declares types and functions that provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[293] A *floating-point status flag* is a system variable whose value is set (but never cleared) when a *floating-point exception* is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information.[294] A *floating-point control mode* is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

2 The floating-point environment has thread storage duration. The initial state for a thread's floating-point environment is the current state of the floating-point environment of the thread that creates it at the time of creation. The floating-point environment is represented by the placeholder identifier **`fenv`** for core function attributes; function synposis are annotated accordingly. Additionally all the functions of this clause are idempotent and a corresponding pragma

```
#pragma CORE FUNCTION_ATTRIBUTE core::idempotent
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(fenv)
```

is implied for the whole header.

3 Certain programming conventions support the intended model of use for the floating-point environment:[295]

— a function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented;

— a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;

— a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

4 The feature test macro `__STDC_VERSION_FENV_H__` expands to the token `202005L`.

5 The complete opaque object type

```
fenv_t
```

represents the entire floating-point environment.

6 The complete opaque object type

```
fexcept_t
```

represents the floating-point status flags collectively, including any status the implementation associates with the flags.

7 Each of the macros

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

[293] This header is designed to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. It is also designed to facilitate code portability among all systems.

[294] A floating-point status flag is not an object and can be set more than once within an expression.

[295] With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

is defined if and only if the implementation supports the floating-point exception by means of the functions in 7.6.2.[296] Additional implementation-defined floating-point exceptions, with macro definitions beginning with **FE_** and an uppercase letter,[297] may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values, and furthermore, bitwise ANDs of all combinations of the macros result in zero.[298]

8 The macro

```
FE_ALL_EXCEPT
```

is simply the bitwise OR of all floating-point exception macros defined by the implementation. If no such macros are defined, **FE_ALL_EXCEPT** shall be defined as `0`.

9 Each of the macros

```
FE_DOWNWARD
FE_TONEAREST
FE_TONEARESTFROMZERO
FE_TOWARDZERO
FE_UPWARD
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the **fegetround** and **fesetround** functions. Additional implementation-defined rounding directions, with macro definitions beginning with **FE_** and an uppercase letter,[299] may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.[300]

10 The macro

```
FE_DFL_ENV
```

represents the default floating-point environment — the one installed at program startup — and has type "pointer to const-qualified **fenv_t**". It can be used as an argument to `<fenv.h>` functions that manage the floating-point environment.

11 Additional implementation-defined environments, with macro definitions beginning with **FE_** and an uppercase letter,[301] and having type "pointer to const-qualified **fenv_t**", may also be specified by the implementation.

7.6.1 The FENV_ACCESS pragma

Synopsis

1
```
#include <fenv.h>
#pragma STDC FENV_ACCESS on-off-switch
```

Description

2 The **FENV_ACCESS** pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.[302] The pragma shall occur either outside external declarations or

[296] The implementation supports a floating-point exception if there are circumstances where a call to at least one of the functions in 7.6.2, using the macro as the appropriate argument, will succeed. It is not necessary for all the functions to succeed all the time.

[297] See "future library directions" (7.32.3).

[298] The macros are typically distinct powers of two.

[299] See "future library directions" (7.32.3).

[300] Even though the rounding direction macros might expand to constants corresponding to the values of **FLT_ROUNDS**, they are not required to do so.

[301] See "future library directions" (7.32.3).

[302] The purpose of the **FENV_ACCESS** pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of **FENV_ACCESS**

preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma "off", the behavior is undefined. The default state ("on" or "off") for the pragma is implementation-defined. (When execution passes from a part of the program translated with **FENV_ACCESS** "off" to a part translated with **FENV_ACCESS** "on", the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

3 **EXAMPLE**

```
#include <fenv.h>
void f(double x)
{
      #pragma STDC FENV_ACCESS ON
      void g(double);
      void h(double);
      /* ... */
      g(x + 1);
      h(x + 1);
      /* ... */
}
```

4 If the function `g` might depend on status flags set as a side effect of the first `x + 1`, or if the second `x + 1` might depend on control modes set as a side effect of the call to function `g`, then the program has to contain an appropriately placed invocation of `#pragma STDC FENV_ACCESS ON` as shown.[303)]

7.6.2 Floating-point exceptions

1 The following functions provide access to the floating-point status flags.[304)] The **int** input argument for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise OR of one or more floating-point exception macros, for example `FE_OVERFLOW | FE_INEXACT`. For other argument values, the behavior of these functions is undefined.

7.6.2.1 The `feclearexcept` function

Synopsis

1
```
#include <fenv.h>
~~int feclearexcept(int excepts);~~
int feclearexcept(int excepts)
      [[ core::modifies(fenv) ]] ;
```

Description

2 The **feclearexcept** function attempts to clear the supported floating-point exceptions represented by its argument.

Returns

3 The **feclearexcept** function returns zero if the `excepts` argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.

is "off", the translator can assume that default modes are in effect and the flags are not tested.

303)The side effects impose a temporal ordering that requires two evaluations of `x + 1`. On the other hand, without the `#pragma STDC FENV_ACCESS ON` pragma, and assuming the default state is "off", just one evaluation of `x + 1` would suffice.

304)The functions **fetestexcept**, **feraiseexcept**, and **feclearexcept** support the basic abstraction of flags that are either set or clear. An implementation can endow floating-point status flags with more information — for example, the address of the code which first raised the floating-point exception; the functions **fegetexceptflag** and **fesetexceptflag** deal with the full content of flags.

7.6.2.2 The fegetexceptflag function

Synopsis

1
```
#include <fenv.h>
int fegetexceptflag(fexcept_t *flagp, int excepts);
int fegetexceptflag(fexcept_t * [[core::writethrough]] flagp, int excepts);
```

Description

2 The **fegetexceptflag** function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by the argument `flagp`.

Returns

3 The **fegetexceptflag** function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.

7.6.2.3 The feraiseexcept function

Synopsis

1
```
#include <fenv.h>
int feraiseexcept(int excepts);
int feraiseexcept(int excepts) [[core::modifies(fenv)]];
```

Description

2 The **feraiseexcept** function attempts to raise the supported floating-point exceptions represented by its argument.[305] The order in which these floating-point exceptions are raised is unspecified, except as stated in F.8.6. Whether the **feraiseexcept** function additionally raises the "inexact" floating-point exception whenever it raises the "overflow" or "underflow" floating-point exception is implementation-defined.

Returns

3 The **feraiseexcept** function returns zero if the `excepts` argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.

7.6.2.4 The fesetexceptflag function

Synopsis

1
```
#include <fenv.h>
int fesetexceptflag(const fexcept_t *flagp,
          int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts)
          [[core::modifies(fenv)]];
```

Description

2 The **fesetexceptflag** function attempts to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to **fegetexceptflag** whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. This function does not raise floating-point exceptions, but only sets the state of the flags.

Returns

3 The **fesetexceptflag** function returns zero if the `excepts` argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.

7.6.2.5 The fetestexcept function

[305] The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken. The specification in F.8.6 is in the same spirit.

Synopsis

1
```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

2 The `fetestexcept` function determines which of a specified subset of the floating-point exception flags are currently set. The `excepts` argument specifies the floating-point status flags to be queried.[306)]

Returns

3 The `fetestexcept` function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.

4 EXAMPLE Call `f` if "invalid" is set, then `g` if "overflow" is set:

```
#include <fenv.h>
/* ... */
{
        #pragma STDC FENV_ACCESS ON
        int set_excepts;
        feclearexcept(FE_INVALID | FE_OVERFLOW);
        // maybe raise exceptions
        set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
        if (set_excepts & FE_INVALID) f();
        if (set_excepts & FE_OVERFLOW) g();
        /* ... */
}
```

7.6.3 Rounding

1 The `fegetround` and `fesetround` functions provide control of rounding direction modes.

7.6.3.1 The `fegetround` function

Synopsis

1
```
#include <fenv.h>
int fegetround(void);
```

Description

2 The `fegetround` function gets the current rounding direction.

Returns

3 The `fegetround` function returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

7.6.3.2 The `fesetround` function

Synopsis

1
```
#include <fenv.h>
int fesetround(int round)
        [[ core::modifies(fenv) ]] ;
```

Description

2 The `fesetround` function establishes the rounding direction represented by its argument `round`. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

306) This mechanism allows testing several floating-point exceptions with just one function call.

Returns

3 The **fesetround** function returns zero if and only if the requested rounding direction was established.

4 **EXAMPLE** Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>

void f(int round_dir)
{
        #pragma STDC FENV_ACCESS ON
        int save_round;
        int setround_ok;
        save_round = fegetround();
        setround_ok = fesetround(round_dir);
        assert(setround_ok == 0);
        assert(setround_ok == 0);
        /* ... */
        fesetround(save_round);
        /* ... */
}
```

7.6.4 Environment

1 The functions in this section manage the floating-point environment — status flags and control modes — as one entity.

7.6.4.1 The fegetenv function

Synopsis

1
```
#include <fenv.h>
int fegetenv(fenv_t *envp);
int fegetenv(fenv_t * [[ core::writethrough ]] envp);
```

Description

2 The **fegetenv** function attempts to store the current floating-point environment in the object pointed to by envp.

Returns

3 The **fegetenv** function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.

7.6.4.2 The feholdexcept function

Synopsis

1
```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
int feholdexcept(fenv_t *envp)
          [[ core::modifies(fenv) ]] ;
```

Description

2 The **feholdexcept** function saves the current floating-point environment in the object pointed to by envp, clears the floating-point status flags, and then installs a *non-stop* (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.[307)]

[307)] IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the **feholdexcept** function can be used in conjunction with the **feupdateenv** function to write routines that hide spurious floating-point exceptions from their callers.

Returns

3 The **feholdexcept** function returns zero if and only if non-stop floating-point exception handling was successfully installed.

7.6.4.3 The fesetenv function

Synopsis

1
```
#include <fenv.h>
~~int fesetenv(const fenv_t *envp);~~
int fesetenv(const fenv_t *envp)
          [[ core::modifies(fenv) ]] ;
```

Description

2 The **fesetenv** function attempts to establish the floating-point environment represented by the object pointed to by envp. The argument envp shall point to an object set by a call to **fegetenv** or **feholdexcept**, or equal a floating-point environment macro. Note that **fesetenv** merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.

Returns

3 The **fesetenv** function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.

7.6.4.4 The feupdateenv function

Synopsis

1
```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
          [[ core::modifies(fenv) ]] ;
```

Description

2 The **feupdateenv** function attempts to save the currently raised floating-point exceptions in its automatic storage, install the floating-point environment represented by the object pointed to by envp, and then raise the saved floating-point exceptions. The argument envp shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a floating-point environment macro.

Returns

3 The **feupdateenv** function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.

4 **EXAMPLE** Hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
{
      #pragma STDC FENV_ACCESS ON
      double result;
      fenv_t save_env;
      if (feholdexcept(&save_env))
            return /* indication of an environmental problem */;
      // compute result
      if (/* test spurious underflow */)
            if (feclearexcept(FE_UNDERFLOW))
                  return /* indication of an environmental problem */;
      if (feupdateenv(&save_env))
            return /* indication of an environmental problem */;
      return result;
}
```

7.7 Characteristics of floating types `<float.h>`

1 The header `<float.h>` defines several macros that expand to various limits and parameters of the standard floating-point types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.2. A summary is given in Annex E.

7.8 Format conversion of integer types `<inttypes.h>`

1 The header `<inttypes.h>` includes the header `<stdint.h>` and extends it with additional facilities provided by hosted implementations.

2 It declares some functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it ~~declares the type which is a structure type that is the type of the value returned by the function.~~ reserves the obsolescent identifier **imaxdiv** for external linkage. For each type declared in `<stdint.h>`, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[308)]

3 The feature test macro **__CORE_VERSION_INTTYPES_H__** expands to the token *202005*L.

Forward references: integer types `<stdint.h>` (7.20), formatted input/output functions (7.21.6), formatted wide character input/output functions (7.29.2).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of **PRI** (character string literals for the **fprintf** and **fwprintf** family) or **SCN** (character string literals for the **fscanf** and **fwscanf** family),[309)] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.20.1. In these names, *N* represents the width of the type as described in 7.20.1. For example, **PRIdFAST32** can be used in a format string to print the value of an integer of type **int_fast32_t**.

2 The **fprintf** macros for signed integers are:

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR

3 The **fprintf** macros for unsigned integers are:

PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR

4 The **fscanf** macros for signed integers are:

SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR

5 The **fscanf** macros for unsigned integers are:

SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

6 For each type that the implementation provides in `<stdint.h>`, the corresponding **fprintf** macros shall be defined and the corresponding **fscanf** macros shall be defined unless the implementation does not have a suitable **fscanf** length modifier for the type.

7 **EXAMPLE**

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
      uintmax_t i = UINTMAX_MAX;    // this type always exists
      wprintf(L"The largest integer value is %020"
            PRIxMAX "\n", i);
      return 0;
```

308) See "future library directions" (7.32.4).

309) Separate macros are given for use with **fprintf** and **fscanf** functions because, in the general case, different format specifiers might be required for **fprintf** and **fscanf**, even when the type is the same.

```
    }
```

7.8.2 Functions for greatest-width integer types

~~Synopsis replace~~
~~**Description**~~

~~The computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.~~

~~**Returns**~~

1 ~~The returns the absolute value.~~

This clause has been removed from the common C/C++ core specification. ~~Synopsis replace~~
~~**Description**~~

~~The computes `numer / denom` and `numer % denom` in a single operation.~~

~~**Returns**~~

The ~~returns a structure of type **`imaxdiv_t`** comprising both the quotient and the remainder. The structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type **`intmax_t`**. If either part of the result cannot be represented, the behavior is undefined. Synopsis replace~~
~~**Description**~~

~~The **`strtoimax`** and **`strtoumax`** functions are equivalent to the , , , and functions, except that the initial portion of the string is converted to **`intmax_t`** and **`uintmax_t`** representation, respectively.~~

~~**Returns**~~

~~The **`strtoimax`** and **`strtoumax`** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, , , or is returned (according to the return type and sign of the value, if any), and the value of the macro is stored in . the **`strtol`**, **`strtoll`**, **`strtoul`**, and **`strtoull`** functions (??). Synopsis replace~~
~~**Description**~~

~~The **`wcstoimax`** and **`wcstoumax`** functions are equivalent to the , , , and functions except that the initial portion of the wide string is converted to~~ following names are used by the C standard for functions in connection with the **`intmax_t`** and **`uintmax_t`** ~~representation, respectively.~~

~~**Returns**~~types:

~~The **`wcstoimax`** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, , , or is returned (according to the return type and sign of the value, if any), and the value of the macro is stored in . the **`wcstol`**, **`wcstoll`**, **`wcstoul`**, and **`wcstoull`** functions (??).~~

`imaxabs`	`imaxabs`	`strtoimax`
`wcstoimax`	`imaxdiv`	`strtoumax`

(no diff marks, here)

7.9 Alternative spellings `<iso646.h>`

1 The obsolete header `<iso646.h>` contains no definitions.

7.10 Characteristics of integer types `<limits.h>`

1 The header `<limits.h>` defines several macros that expand to various limits and parameters of the standard integer types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.1. A summary is given in Annex E.

7.11 Localization <locale.h>

1 The header `<locale.h>` declares two functions, one type, and defines several macros.

2 The type is

```
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained in 7.11.2.1. In the `"C"` locale, the members shall have the values specified in the comments.

```
char *decimal_point;       // "."
char *thousands_sep;       // ""
char *grouping;            // ""
char *mon_decimal_point;   // ""
char *mon_thousands_sep;   // ""
char *mon_grouping;        // ""
char *positive_sign;       // ""
char *negative_sign;       // ""
char *currency_symbol;     // ""
char frac_digits;          // CHAR_MAX
char p_cs_precedes;        // CHAR_MAX
char n_cs_precedes;        // CHAR_MAX
char p_sep_by_space;       // CHAR_MAX
char n_sep_by_space;       // CHAR_MAX
char p_sign_posn;          // CHAR_MAX
char n_sign_posn;          // CHAR_MAX
char *int_curr_symbol;     // ""
char int_frac_digits;      // CHAR_MAX
char int_p_cs_precedes;    // CHAR_MAX
char int_n_cs_precedes;    // CHAR_MAX
char int_p_sep_by_space;   // CHAR_MAX
char int_n_sep_by_space;   // CHAR_MAX
char int_p_sign_posn;      // CHAR_MAX
char int_n_sign_posn;      // CHAR_MAX
```

3 The macros defined are

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the `setlocale` function.[310] Additional macro definitions, beginning with the characters `LC_` and an uppercase letter,[311] may also be specified by the implementation.

4 The locale functions access a hidden state `locale` as do many other library functions that rely on specific locale information.

Recommended practice

5 It is recommended that all implementation specific function declarations that depend on locale

[310] ISO/IEC 9945–2 specifies locale and charmap formats that can be used to specify locales for C.
[311] See "future library directions" (7.32.5).

information are annotated with the appropriate core function attributes.

7.11.1 Locale control

7.11.1.1 The setlocale function

Synopsis

1
```
#include <locale.h>
char *setlocale(int category, const char *locale);
char *setlocale(int category, const char *locale)
        [[core::unsequenced, core::modifies(locale)]];
```

Description

2 The **setlocale** function selects the appropriate portion of the program's locale as specified by the category and **locale** arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for category names the program's entire locale; the other values for category name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions[312] and the multibyte and wide character functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** and **wcsftime** functions.

3 A value of "C" for **locale** specifies the minimal environment for C translation; a value of "" for **locale** specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

4 At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

5 A call to the **setlocale** function may introduce a data race with other calls to the **setlocale** function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the **setlocale** function.

Returns

6 If a pointer to a string is given for **locale** and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

7 A null pointer for **locale** causes the **setlocale** function to return a pointer to the string associated with the category for the program's current locale; the program's locale is not changed.[313]

8 The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Forward references: formatted input/output functions (7.21.6), multibyte/wide character conversion functions (7.22.8), multibyte/wide string conversion functions (7.22.9), numeric conversion functions (7.22.1), the **strcoll** ~~function (??~~type-generic macro (7.24.4.3), the **strftime** function (7.27.3.5), the **strxfrm** ~~function (??).~~ type-generic macro (7.24.4.5).

[312]The only functions in 7.4 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.
[313]The implementation is thus required to arrange to encode in a string the various categories due to a heterogeneous locale when category has the value **LC_ALL**.

7.11.2 Numeric formatting convention inquiry

7.11.2.1 The `localeconv` function

Synopsis

1
```
#include <locale.h>
~~struct lconv *localeconv(void);~~
struct lconv *localeconv(void) [[unsequenced, core::evaluates(locale)]];
```

Description

2 The `localeconv` function sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

3 The members of the structure with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. Apart from `grouping` and `mon_grouping`, the strings shall start and end in the initial shift state. The members with type `char` are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The members include the following:

`char *decimal_point`
: The decimal-point character used to format nonmonetary quantities.

`char *thousands_sep`
: The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

`char *grouping`
: A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

`char *mon_decimal_point`
: The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
: The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
: A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`
: The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
: The string used to indicate a negative-valued formatted monetary quantity.

`char *currency_symbol`
: The local currency symbol applicable to the current locale.

`char frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

`char p_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

`char n_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

`char p_sep_by_space`
: Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a nonnegative locally formatted monetary quantity.

`char n_sep_by_space`
: Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative locally formatted monetary quantity.

`char p_sign_posn`
: Set to a value indicating the positioning of the `positive_sign` for a nonnegative locally formatted monetary quantity.

`char n_sign_posn`
: Set to a value indicating the positioning of the `negative_sign` for a negative locally formatted monetary quantity.

`char *int_curr_symbol`
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

`char int_frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

`char int_p_cs_precedes`
: Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

`char int_n_cs_precedes`
: Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

`char int_p_sep_by_space`
: Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

`char int_n_sep_by_space`
: Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a negative internationally formatted monetary quantity.

`char int_p_sign_posn`
: Set to a value indicating the positioning of the `positive_sign` for a nonnegative internationally formatted monetary quantity.

`char int_n_sign_posn`
: Set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity.

4 The elements of `grouping` and `mon_grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping is to be performed.

`0` The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

5 The values of **`p_sep_by_space`**, **`n_sep_by_space`**, **`int_p_sep_by_space`**, and **`int_n_sep_by_space`** are interpreted according to the following:

`0` No space separates the currency symbol and value.

`1` If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

`2` If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For **`int_p_sep_by_space`** and **`int_n_sep_by_space`**, the fourth character of **`int_curr_symbol`** is used instead of a space.

6 The values of **`p_sign_posn`**, **`n_sign_posn`**, **`int_p_sign_posn`**, and **`int_n_sign_posn`** are interpreted according to the following:

`0` Parentheses surround the quantity and currency symbol.

`1` The sign string precedes the quantity and currency symbol.

`2` The sign string succeeds the quantity and currency symbol.

`3` The sign string immediately precedes the currency symbol.

`4` The sign string immediately succeeds the currency symbol.

7 The implementation shall behave as if no library function calls the **`localeconv`** function.

Returns

8 The **`localeconv`** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **`localeconv`** function. In addition, calls to the **`setlocale`** function with categories **`LC_ALL`**, **`LC_MONETARY`**, or **`LC_NUMERIC`** may overwrite the contents of the structure.

9 **EXAMPLE 1** The following table illustrates rules which might well be used by four countries to format monetary quantities.

	Local format		International format	
Country	Positive	Negative	Positive	Negative
Country1	`1.234,56 mk`	`-1.234,56 mk`	`FIM 1.234,56`	`FIM -1.234,56`
Country2	`L.1.234`	`-L.1.234`	`ITL 1.234`	`-ITL 1.234`
Country3	`ƒ1.234,56`	`ƒ-1.234,56`	`NLG 1.234,56`	`NLG -1.234,56`
Country4	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`	`CHF 1,234.56C`

10 For these four countries, the respective values for the monetary members of the structure returned by **`localeconv`** could be:

	Country1	Country2	Country3	Country4
mon_decimal_point	","	""	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
currency_symbol	"mk"	"L."	"\u0192"	"SFrs."
frac_digits	2	0	2	2
p_cs_precedes	0	1	1	1
n_cs_precedes	0	1	1	1
p_sep_by_space	1	0	1	0
n_sep_by_space	1	0	2	0
p_sign_posn	1	1	1	1
n_sign_posn	1	1	4	2
int_curr_symbol	"FIM "	"ITL "	"NLG "	"CHF "
int_frac_digits	2	0	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	1	1	1	1
int_n_sep_by_space	2	1	2	1
int_p_sign_posn	1	1	1	1
int_n_sign_posn	4	1	4	2

11 **EXAMPLE 2** The following table illustrates how the `cs_precedes`, `sep_by_space`, and `sign_posn` members affect the formatted value.

		p_sep_by_space		
p_cs_precedes	p_sign_posn	0	1	2
0	0	(1.25$)	(1.25 $)	(1.25$)
	1	+1.25$	+1.25 $	+ 1.25$
	2	1.25$+	1.25 $+	1.25$ +
	3	1.25+$	1.25 +$	1.25+ $
	4	1.25$+	1.25 $+	1.25$ +
1	0	($1.25)	($ 1.25)	($1.25)
	1	+$1.25	+$ 1.25	+ $1.25
	2	$1.25+	$ 1.25+	$1.25 +
	3	+$1.25	+$ 1.25	+ $1.25
	4	$+1.25	$+ 1.25	$ +1.25

7.12 Mathematics `<math.h>`

1 The header `<math.h>` declares two types~~and many mathematical functions and defines several macros. Most synopses specify a family of functions~~, and many type-generic macros and some functions to provide interfaces for mathematical functions. Additionally, many obsolecent function names are also reserved, consisting of a principal function (of the same name as the type-generic macro) with one or more **`double`** parameters, a **`double`** return value, or both; and other functions with the same name but with **`f`** and **`l`** suffixes, which are corresponding functions with **`float`** and **`long double`** parameters, return values, or both.[314] The identifiers that are such reserved are:

acosf	cosl	fmin	logbf	rintl
acoshf	cos	fmodf	logbl	rint
acoshl	erfcf	fmodl	logb	roundf
acosh	erfcl	fmod	logf	roundl
acosl	erfc	frexpf	logl	round
acos	erff	frexpl	log	scalblnf
asinf	erfl	frexp	lrintf	scalblnl
asinhf	erf	hypotf	lrintl	scalbln
asinhl	exp2f	hypotl	lrint	scalbnf
asinh	exp2l	hypot	lroundf	scalbnl
asinl	exp2	ilogbf	lroundl	scalbn
asin	expf	ilogbl	lround	sinf
atan2f	expl	ilogb	nearbyintf	sinhf
atan2l	expm1f	ldexpf	nearbyintl	sinhl
atan2	expm1l	ldexpl	nearbyint	sinh
atanf	expm1	ldexp	nextafterf	sinl
atanhf	exp	lgammaf	nextafterl	sin
atanhl	fabsf	lgammal	nextafter	sqrtf
atanh	fabsl	lgamma	nexttowardf	sqrtl
atanl	fabs	llrintf	nexttowardl	sqrt
atan	fdimf	llrintl	nexttoward	tanf
cbrtf	fdiml	llrint	modff	tanhf
cbrtl	fdim	llroundf	modfl	tanhl
cbrt	floorf	llroundl	modf	tanh
ceilf	floorl	llround	powf	tanl
ceill	floor	log10f	powl	tan
ceil	fmaf	log10l	pow	tgammaf
copysignf	fmal	log10	remainderf	tgammal
copysignl	fmaxf	log1pf	remainderl	tgamma
copysign	fmaxl	log1pl	remainder	truncf
cosf	fmax	log1p	remquof	truncl
coshf	fma	log2f	remquol	trunc
coshl	fminf	log2l	remquo	
cosh	fminl	log2	rintf	

2 For the synopsis of the type-generic macros, R, S and T denote real types that are used to describe the underspecified parameter types. If F denotes the inferred return type, it is a floating type: if the underspecified argument types to the call are all integer type, F is **`double`**. Otherwise, F is the common type of the underspecified argument types as inferred by usual arithmetic conversion. The effect is then as if a function where R, S and T are F is called and the arguments are converted accordingly. If the inferred return type is specified with another letter than F, the description of the corresponding clause gives the details.

3 The provisions of this clause not withstanding, unless the macro `__CORE_NO_COMPLEX__` is defined, several of the type-generic macros can be amended by the inclusion of the `<complex.h>` for

[314] Particularly on systems with wide expression evaluation, a `<math.h>` function might pass arguments and return values in wider format than the synopsis prototype indicates.

complex types.

4 Such a type-generic macro can be used outside of an actual function call for a conversion to a function pointer of type where the underspecified parameter types *R*, *S* and *T* are all fixed to the same type as either **float**, **double** or **long double** and where the return type is as described in the synopsis.[315]

5 Integer arithmetic functions and conversion functions are discussed later.

6 Attributes corresponding to the pragmas

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced
#pragma CORE FUNCTION_ATTRIBUTE core::modifies(errno, fenv)
```

are be implied for the whole header, only that an implementation may strengthen the **core::modifies** attribute to one or zero of the identifiers if it can guarantee that the corresponding C library channel is not affected by the function.[316]

7 The feature test macro ~~__STDC_VERSION_MATH_H__~~ __CORE_VERSION_MATH_H__ expands to the token *202005*L.

8 The types

```
float_t
double_t
```

are floating types at least as wide as **float** and **double**, respectively, and such that **double_t** is at least as wide as **float_t**. If **FLT_EVAL_METHOD** equals 0, **float_t** and **double_t** are **float** and **double**, respectively; if **FLT_EVAL_METHOD** equals 1, they are both **double**; if **FLT_EVAL_METHOD** equals 2, they are both **long double**; and for other values of **FLT_EVAL_METHOD**, they are otherwise implementation-defined.[317]

9 The macro

```
HUGE_VAL
```

expands to a positive **double** constant expression, not necessarily representable as a **float**. The macros

```
HUGE_VALF
HUGE_VALL
```

are respectively **float** and **long double** analogs of **HUGE_VAL**.[318]

10 The macro

```
INFINITY
```

expands to a constant expression of type **float** representing positive or unsigned infinity, if available; else to a positive constant of type **float** that overflows at translation time.[319]

11 The macro

[315]For example the **frexp** macro can be converted to function pointer types `R(*)(R, int*)` for any floating point type `R`. One possibility to ensure such a conversion is to implement the type-generic macro as a generic lambda expression.

[316]That means that translators may move all calls to the type-generic macros as early as their arguments are available, that the changes to the state other than the return value are idempotent, restricted to **errno** and the floating-point state, and that these changes only depend on the arguments to the call.

[317]The types **float_t** and **double_t** are intended to be the implementation's most efficient types at least as wide as **float** and **double**, respectively. For **FLT_EVAL_METHOD** equal 0, 1, or 2, the type **float_t** is the narrowest type used by the implementation to evaluate floating expressions.

[318]**HUGE_VAL**, **HUGE_VALF**, and **HUGE_VALL** can be positive infinities in an implementation that supports infinities.

[319]In this case, using **INFINITY** will violate the constraint in 6.4.4 and thus require a diagnostic.

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet NaN.

12 The *number classification macros*

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with **FP_** and an uppercase letter, may also be specified by the implementation.

13 The macro

```
FP_FAST_FMA
```

is optionally defined. If defined, it indicates that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of **double** operands.[320] The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, **float** and **long double** analogs of **FP_FAST_FMA**. If defined, these macros expand to the integer constant 1.

14 The macros

```
FP_ILOGB0
FP_ILOGBNAN
```

expand to integer constant expressions whose values are returned by **ilogb**(x) if x is zero or NaN, respectively. The value of **FP_ILOGB0** shall be either **INT_MIN** or -**INT_MAX**. The value of **FP_ILOGBNAN** shall be either **INT_MAX** or **INT_MIN**.

15 The macros

```
MATH_ERRNO
MATH_ERREXCEPT
```

expand to the integer constants 1 and 2, respectively; the macro

```
math_errhandling
```

expands to an expression that has type **int** and the value **MATH_ERRNO**, **MATH_ERREXCEPT**, or the bitwise OR of both. The value of **math_errhandling** is constant for the duration of the program. It is unspecified whether **math_errhandling** is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name **math_errhandling**, the behavior is undefined. If the expression **math_errhandling** & **MATH_ERREXCEPT** can be nonzero, the implementation shall define the macros **FE_DIVBYZERO**, **FE_INVALID**, and **FE_OVERFLOW** in <fenv.h>.

[320]Typically, the **FP_FAST_FMA** macro is defined if and only if the **fma** function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.

7.12.1 Treatment of error conditions

1 The behavior of each of the functions in `<math.h>` is specified for all representable values of its input arguments, except where explicitly stated otherwise. Each function shall execute as if it were a single operation without raising **`SIGFPE`** and without generating any of the floating-point exceptions "invalid", "divide-by-zero", or "overflow" except to reflect the result of the function.

2 For all functions, a *domain error* occurs if and only if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[321] On a domain error, the function returns an implementation-defined value; if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, the integer expression **`errno`** acquires the value **`EDOM`**; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, the "invalid" floating-point exception is raised.

3 Similarly, a *pole error* (also known as a singularity or infinitary) occurs if and only if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit (for example, **`log`**`(0.0)`). The description of each function lists any required pole errors; an implementation may define additional pole errors, provided that such errors are consistent with the mathematical definition of the function. On a pole error, the function returns an implementation-defined value; if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, the integer expression **`errno`** acquires the value **`ERANGE`**; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, the "divide-by-zero" floating-point exception is raised.

4 Likewise, a *range error* occurs if and only if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude. The description of each function lists any required range errors; an implementation may define additional range errors, provided that such errors are consistent with the mathematical definition of the function and are the result of either overflow or underflow.

5 A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the function returns the value of the macro **`HUGE_VAL`**, **`HUGE_VALF`**, or **`HUGE_VALL`** according to the return type, with the same sign as the correct value of the function; if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, the integer expression **`errno`** acquires the value **`ERANGE`**; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, the "overflow" floating-point exception is raised.

6 The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.[322] If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, whether **`errno`** acquires the value **`ERANGE`** is implementation-defined; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

7 If a domain, pole, or range error occurs and the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is zero,[323] then **`errno`** shall either be set to the value corresponding to the error or left unmodified. If no such error occurs, **`errno`** shall be left unmodified regardless of the setting of **`math_errhandling`**.

8 For the functions and macros below that have a **`constexpr`** specifier, such error conditions that occur during the evaluation of a constant expression shall be diagnosed and shall be considered as a failure condition for the evaluation of the constant expression.[324]

[321] In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

[322] The term underflow here is intended to encompass both "gradual underflow" as in IEC 60559 and also "flush-to-zero" underflow.

[323] Math errors are being indicated by the floating-point exception flags rather than by **`errno`**.

[324] A C library implementation may communicate such error conditions by raising the appropriate floating point conditions

7.12.2 The `FP_CONTRACT` pragma

Synopsis

1
```
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

Description

2 The `FP_CONTRACT` pragma can be used to allow (if the state is "on") or disallow (if the state is "off") the implementation to contract expressions (6.5). Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state ("on" or "off") for the pragma is implementation-defined.

7.12.3 Classification macros

Constraints

1 In the synopses ~~in~~ of this subclause, ~~indicates that the argument shall be an expression of real floating~~ *R* shall be a real floating point argument type.

Description

2 Outside a function call, these macros can be converted to function pointer types **int**(∗)(R) or **bool**(∗)(R), respectively, where R is a floating point type.

7.12.3.1 The `fpclassify` macro

Synopsis

1
```
#include <math.h>
int fpclassify(real-floating x);
constexpr int fpclassify(R x);
```

Description

2 The `fpclassify` macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[325)]

Returns

3 The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

7.12.3.2 The `isfinite` macro

Synopsis

1
```
#include <math.h>
int isfinite(real-floating x);
constexpr bool isfinite(R x);
```

---

or by setting **errno**, or it may use any other specific way to conway that information such that the error condition is diagnosed during compilation.

325)Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal **long double** value might become subnormal when converted to **double**, and zero when converted to **float**.

Description

2 The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **isfinite** macro returns ~~a nonzero value~~ **true** if and only if its argument has a finite value.

7.12.3.3 The isinf macro

Synopsis

1
```
#include <math.h>
int isinf(real-floating x);
constexpr bool isinf(R x);
```

Description

2 The **isinf** macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **isinf** macro returns ~~a nonzero value~~ **true** if and only if its argument has an infinite value.

7.12.3.4 The isnan macro

Synopsis

1
```
#include <math.h>
int isnan(real-floating x);
constexpr bool isnan(R x);
```

Description

2 The **isnan** macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.[326]

Returns

3 The **isnan** macro returns ~~a nonzero value~~ **true** if and only if its argument has a NaN value.

7.12.3.5 The isnormal macro

Synopsis

1
```
#include <math.h>
int isnormal(real-floating x);
constexpr bool isnormal(R x);
```

Description

2 The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **isnormal** macro returns ~~a nonzero value~~ **true** if and only if its argument has a normal value.

7.12.3.6 The signbit macro

[326] For the **isnan** macro , the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

Synopsis

1
```
#include <math.h>
int signbit(real-floating x);
constexpr bool signbit(R x);
```

Description

2 The **signbit** macro determines whether the sign of its argument value is negative.[327)]

Returns

3 The **signbit** macro returns ~~a nonzero value~~ **true** if and only if the sign of its argument value is negative.

7.12.4 Trigonometric functions

7.12.4.1 The acos type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F acos(R x);
```

Description

2 The **acos** type-generic macro ~~compute~~ computes the principal value of the arc cosine of x. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **acos** type-generic macro ~~return~~ returns $\arccos$ x in the interval $[0, \pi]$ radians.

7.12.4.2 The asin type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F asin(R x);
```

Description

2 The **asin** type-generic macro ~~compute~~ computes the principal value of the arc sine of x. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **asin** type-generic macro ~~return~~ returns $\arcsin$ x in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

7.12.4.3 The atan type-generic macro

~~Synopsis replace~~

327) The **signbit** macro reports the sign of all values, including infinities, zeros, and NaNs. If zero is unsigned, it is treated as positive.

```
#include <math.h>
double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F atan(R x);
```

Description

2 The **atan** type-generic macro ~~compute~~ computes the principal value of the arc tangent of x.

Returns

3 The **atan** type-generic macro ~~return~~ returns $\arctan x$ in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

7.12.4.4 The atan2 type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F atan2(R x, S y);
```

Description

2 The **atan2** type-generic macro ~~compute~~ computes the value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns

3 The **atan2** type-generic macro ~~return~~ returns $\arctan y/x$ in the interval $[-\pi, +\pi]$ radians.

7.12.4.5 The cos type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F cos(R x);
```

Description

2 The **cos** type-generic macro ~~compute~~ computes the cosine of x (measured in radians).

Returns

3 The **cos** type-generic macro ~~return~~ returns $\cos x$.

7.12.4.6 The sin type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F sin(R x);
```

Description

2 The **sin** type-generic macro ~~compute~~ computes the sine of x (measured in radians).

Returns

3 The **sin** type-generic macro ~~return~~ returns $\sin$ x.

7.12.4.7 The tan type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F tan(R x);
```

Description

2 The **tan** type-generic macro ~~return~~ returns the tangent of x (measured in radians).

Returns

3 The **tan** type-generic macro ~~return~~ returns $\tan$ x.

7.12.5 Hyperbolic functions

7.12.5.1 The acosh type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F acosh(R x);
```

Description

2 The **acosh** type-generic macro ~~compute~~ computes the (nonnegative) arc hyperbolic cosine of x. A domain error occurs for arguments less than 1.

Returns

3 The **acosh** type-generic macro ~~return~~ returns arcosh x in the interval $[0, +\infty]$.

7.12.5.2 The asinh type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F asinh(R x);
```

Description

2 The **asinh** type-generic macro ~~compute~~ computes the arc hyperbolic sine of x.

Returns

3 The **asinh** type-generic macro ~~return~~ returns arsinh x.

7.12.5.3 The atanh type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F atanh(R x);
```

Description

2 The **atanh** type-generic macro ~~compute~~ computes the arc hyperbolic tangent of x. A domain error occurs for arguments not in the interval $[-1, +1]$. A pole error may occur if the argument equals ~~-1 or +1~~ -1 or $+1$.

Returns

3 The **atanh** type-generic macro ~~return~~ returns artanh x.

7.12.5.4 The cosh type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F cosh(R x);
```

Description

2 The **cosh** type-generic macro ~~compute~~ computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Returns

3 The **cosh** type-generic macro ~~return~~ returns cosh x.

7.12.5.5 The **sinh** type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F sinh(R x);
```

Description

2 The **sinh** type-generic macro ~~compute~~ computes the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

Returns

3 The **sinh** type-generic macro ~~return~~ returns sinh x.

7.12.5.6 The **tanh** type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F tanh(R x);
```

Description

2 The **tanh** type-generic macro ~~compute~~ computes the hyperbolic tangent of x.

Returns

3 The **tanh** type-generic macro ~~return~~ returns tanh x.

7.12.6 Exponential and logarithmic functions

7.12.6.1 The **exp** type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double exp(double x);
float expf(float x);
long double expl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F exp(R x);
```

Description

2 The **exp** type-generic macro ~~compute~~ computes the base-e exponential of x. A range error occurs if the magnitude of x is too large.

Returns

3 The **exp** type-generic macro ~~return~~ returns $e^{\mathtt{x}}$.

7.12.6.2 The exp2 type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F exp2(R x);
```

Description

2 The **exp2** type-generic macro ~~compute~~ computes the base-2 exponential of x. A range error occurs if the magnitude of x is too large.

Returns

3 The **exp2** type-generic macro ~~return~~ returns 2^{x}.

7.12.6.3 The expm1 type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F expm1(R x);
```

Description

2 The **expm1** type-generic macro ~~compute~~ computes the base-e exponential of the argument, minus ~~1.~~ 1. A range error occurs if positive x is too large.[328)]

Returns

3 The **expm1** type-generic macro ~~return~~ returns $e^{x} - 1$.

7.12.6.4 The frexp type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
```

Synopsis

1
```
#include <math.h>
F frexp(R x, int * [[core::writethrough]] exp);
```

Description

2 The **frexp** type-generic macro ~~break~~ breaks a floating-point number into a normalized fraction and an integral power of 2. ~~They store~~ It stores the integer in the **int** object pointed to by **exp**.

[328)] For small magnitude x, **expm1**(x) is expected to be more accurate than **exp**(x)-1.

Returns

3 If `value` is not a floating-point number or if the integral power of 2 is outside the range of **`int`**, the results are unspecified. Otherwise, the **`frexp`** type-generic macro ~~return~~ returns the value `x`, such that `x` has a magnitude in the interval $[\frac{1}{2}, 1)$ or zero, and `value` equals `x` $\times$ $2^{*\texttt{exp}}$. If `value` is zero, both parts of the result are zero.

4 NOTE Because it returns one of its results through a pointer paramter, the **`frexp`** type-generic macro is not suited for an evaluation in a constant expression or **`constexpr`** function or lambda. Applications that need the two results in a **`constexpr`** should use combinations of **`log2`**, **`ilogb`** and **`ldexp`** or similar functions to achieve the same results.

7.12.6.5 The `ilogb` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr int ilogb(R x);
```

Description

2 The **`ilogb`** type-generic macro ~~extract~~ extracts the exponent of `x` as a signed **`int`** value. If `x` is zero ~~they compute~~ it computes the value **`FP_ILOGB0`**; if `x` is infinite ~~they compute~~ it computes the value **`INT_MAX`**; if `x` is a NaN ~~they compute~~ it computes the value **`FP_ILOGBNAN`**; otherwise, ~~they are~~ it is equivalent to calling the ~~corresponding~~ **`logb`** ~~function and casting~~ **`logb`** macro and converting the returned value to type **`int`**. A domain error or range error may occur if `x` is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **`ilogb`** type-generic macro ~~return~~ returns the exponent of `x` as a signed **`int`** value.

Forward references: the **`logb`** ~~functions (??~~macro (7.12.6.11).

7.12.6.6 The `ldexp` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Synopsis

1
```
#include <math.h>
constexpr F ldexp(R x, int k);
```

Description

2 The **`ldexp`** type-generic macro ~~multiply~~ multiplies a floating-point number by an integral power of ~~2.~~ 2. A range error may occur.

Returns

3 The **`ldexp`** type-generic macro ~~return `x` $\times$ $2^{\texttt{exp}}$~~returns `x` $\times$ $2^{\texttt{k}}$.

7.12.6.7 The `log` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F log(R x);
```

Description

2 The **log** type-generic macro ~~compute~~ computes the base-e (natural) logarithm of x. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The **log** type-generic macro ~~return~~ returns $\log_e$ x.

7.12.6.8 The log10 type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F log10(R x);
```

Description

2 The **log10** type-generic macro ~~compute~~ computes the base-10 (common) logarithm of x. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The **log10** type-generic macro ~~return~~ returns $\log_{10}$ x.

7.12.6.9 The log1p type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F log1p(R x);
```

Description

2 The **log1p** type-generic macro ~~compute~~ computes the base-e (natural) logarithm of ~~1~~ 1 plus the argument.[329] A domain error occurs if the argument is less than -1. A pole error may occur if the argument equals -1.

Returns

3 The **log1p** type-generic macro ~~return~~ returns $\log_e(1 + \mathtt{x})$.

329) For small magnitude x, **log1p**(x) is expected to be more accurate than **log**(1 + x).

7.12.6.10 The `log2` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Synopsis

1

```
#include <math.h>
constexpr F log2(R x);
```

Description

2 The **`log2`** type-generic macro ~~compute the base-2~~ computes the base-2 logarithm of `x`. A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

3 The **`log2`** type-generic macro ~~return~~ returns $\log_2 x$.

7.12.6.11 The `logb` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Synopsis

1

```
#include <math.h>
constexpr F logb(R x);
```

Description

2 The **`logb`** type-generic macro ~~extract~~ extracts the exponent of `x`, as a signed integer value in floating-point format. If `x` is subnormal it is treated as though it were normalized; thus, for positive finite `x`,

$$1 \le \mathtt{x} \times \mathtt{FLT\_RADIX}^{-\mathtt{logb}(\mathtt{x})} < \mathtt{FLT\_RADIX}$$

A domain error or pole error may occur if the argument is zero.

Returns

3 The **`logb`** type-generic macro ~~return~~ returns the signed exponent of `x`.

7.12.6.12 The `modf` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Synopsis

1

```
#include <math.h>
R modf(Q value, R * [[ core::writethrough ]] iptr);
```

Constraints

2 *R* shall be a non-qualified real floating type.

Description

3 The `modf` type-generic macro ~~break the argument `value`~~ converts the real argument `value` to *R* and breaks the result into integral and fractional parts, each of which has ~~the same type and sign as~~ type *R* and the sign of the argument. ~~They store~~ It stores the integral part (in floating-point format) in the object pointed to by `iptr`.

4 Outside of a function call, the `modf` type-generic macro can be converted to a function pointer of type `R(*)(R, R*)` where `R` is a real floating point type.

Returns

5 The `modf` type-generic macro ~~return~~ returns the signed fractional part of the converted `value`.

~~Synopsis replace~~

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

6 NOTE Because it returns one of its results through a pointer paramter, the `modf` type-generic macro is not suited for an evaluation in a constant expression or `constexpr` function or lambda. Applications that need the two results in a `constexpr` should use combinations of `trunc` or similar functions and arithmetic to achieve the same results.

7.12.6.13 The `scalbn` type-generic macro

Synopsis

1
```
#include <math.h>
constexpr F scalbn(R x, Z n);
```

Constraints

2 *R* shall be a real floating type and *Z* shall be an integer type.

Description

3 The ~~`scalbn` and `scalbln` functions compute~~ `scalbn` type-generic macro computes $\texttt{x} \times \texttt{FLT\_RADIX}^{\texttt{n}}$ efficiently, not normally by computing $\texttt{FLT\_RADIX}^{\texttt{n}}$ explicitly. The value of `n` shall be in the value range of `long int`. A range error may occur.

4 Outside of a function call, the `scalbn` type-generic macro can be converted to a function pointer of type `R(*)(R, Z)` where `R` is a real floating point type and `Z` is `int` or `long int`.

Returns

5 The ~~`scalbn` and `scalbln` functions~~ `scalbn` type-generic macro return $\texttt{x} \times \texttt{FLT\_RADIX}^{\texttt{n}}$.

7.12.7 Power and absolute-value functions

7.12.7.1 The `cbrt` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F cbrt(R x);
```

Description

2 The **cbrt** type-generic macro ~~compute~~ computes the real cube root of x.

Returns

3 The **cbrt** type-generic macro ~~return~~ returns $x^{\frac{1}{3}}$.

7.12.7.2 The fabs type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr R fabs(R x);
```

Description

2 The **fabs** type-generic macro ~~compute~~ computes the absolute value of a floating-point number x. This type-generic macro is not suitable for integer arguments.

Returns

3 The **fabs** type-generic macro ~~return~~ returns $|x|$.

7.12.7.3 The hypot type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F hypot(R x, S y);
```

Description

2 The **hypot** type-generic macro ~~compute~~ computes the square root of the sum of the squares of x and y, without undue overflow or underflow. A range error may occur.

3

Returns

4 The **hypot** type-generic macro ~~return~~ returns $\sqrt{x^2 + y^2}$.

7.12.7.4 The pow type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F pow(R x, S y);
```

Description

2 The **pow** type-generic macro ~~compute~~ computes x raised to the power y. A domain error occurs if x is finite and negative and y is finite and not an integer value. A range error may occur. A domain error may occur if x is zero and y is zero. A domain error or pole error may occur if x is zero and y is less than zero.

Returns

3 The **pow** type-generic macro ~~return~~ returns x^y.

7.12.7.5 The sqrt type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F sqrt(R x);
```

Description

2 The **sqrt** type-generic macro ~~compute~~ computes the nonnegative square root of x. A domain error occurs if the argument is less than zero.

Returns

3 The **sqrt** type-generic macro ~~return~~ returns $\sqrt{x}$.

7.12.7.6 The abs type-generic macro

Synopsis

1
```
#include <math.h>
constexpr U abs(R x);
```

Constraints

2 *R* shall be an arithmetic type.

Description

3 The **abs** type-generic macro computes the absolute value of x. The inferred return type *U* is a real type. If *R* is a narrow integer type, *U* is **unsigned**. Otherwise, if *R* is a real floating point type or an unsigned integer type, *U* is *R*. If *R* is a complex type, *U* is the corresponding real type. Otherwise, R is a wide signed integer type and *U* is the corresponding unsigned type. If *R* is a real type, the mathematical value is always representable exactly in *U*; no error occurs. If *R* is a complex type, **abs**(x) is equivalent to a call **hypot**(**real_value**(x), **imaginary_value**(x)), only that x is evaluated at most once.

4 The **abs** type-generic macro can be converted to a function pointer type R(*)(R) where R is a real floating point type or wide unsigned integer type, to U(*)(R) where R is a complex type and U is the corresponding real type, or to U(*)(R) where R is a wide signed integer type and U is the corresponding unsigned type.

Returns

5 The **abs** type-generic macro returns $|x|$.

6 NOTE Historically, C has **abs** functions for signed types (in <stdlib.h>). They return a signed value such that the absolute value of the minimal value of the type is not representable and thus a call with such a value is undefined. Applications should prefer the type-generic macro here over these legacy interfaces, because here the mathematical result

is always representable in the target type.

7.12.7.7 The **abs**^{2} type-generic macro

Synopsis

1
```
#include <math.h>
constexpr U abs^2(R x);
```

Constraints

2 *R* shall be a floating type.

Description

3 The **abs**^{2} type-generic macro computes the square of the absolute value of `x`. The inferred return type *U* is a real type. If *R* is a real floating point type, *U* is *R*. If *R* is a complex type, *U* is the corresponding real type; **abs**^{2}`(x)` is equivalent to the expression `real_value(x)×real_value(x) + imaginary_value(x)×imaginary_value(x)`, only that `x` is evaluated at most once.

4 The **abs**^{2} type-generic macro can be converted to a function pointer type `R(*)(R)` where `R` is a real floating point type, or to `U(*)(R)` where `R` is a complex type and `U` is the corresponding real type.

Returns

5 The **abs**^{2} type-generic macro returns $|\mathtt{x}|^2$.

6 **NOTE** If the absolute value of complex numbers is only computed to compare the magnitude, **abs**^{2} may be more efficient than to compute **abs** because the computation of the square root is avoided.

7.12.8 Error and gamma functions

7.12.8.1 The **erf** type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F erf(R x);
```

Description

2 The **erf** type-generic macro ~~compute~~ computes the error function of `x`.

Returns

3 The **erf** type-generic macro ~~return~~ returns $\operatorname{erf} \mathtt{x} = \frac{2}{\sqrt{\pi}} \int_0^{\mathtt{x}} e^{-t^2}\, dt$.

7.12.8.2 The **erfc** type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F erfc(R x);
```

Description

2 The **erfc** type-generic macro ~~compute~~ computes the complementary error function of x. A range error occurs if positive x is too large.

Returns

3 The **erfc** type-generic macro ~~return~~ returns $\mathrm{erfc}\,\mathtt{x} = 1 - \mathrm{erf}\,\mathtt{x} = \frac{2}{\sqrt{\pi}} \int_{\mathtt{x}}^{\infty} e^{-t^2}\,dt$.

7.12.8.3 The lgamma type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F lgamma(R x);
```

Description

2 The **lgamma** type-generic macro ~~compute~~ computes the natural logarithm of the absolute value of gamma of x. A range error occurs if positive x is too large. A pole error may occur if x is a negative integer or zero.

Returns

3 The **lgamma** type-generic macro ~~return~~ returns $\log_e |\Gamma(\mathtt{x})|$.

7.12.8.4 The tgamma type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F tgamma(R x);
```

Description

2 The **tgamma** type-generic macro ~~compute~~ computes the gamma function of x. A domain error or pole error may occur if x is a negative integer or zero. A range error occurs if the magnitude of x is too large and may occur if the magnitude of x is too small.

Returns

3 The **tgamma** type-generic macro ~~return~~ returns $\Gamma(\mathtt{x})$.

7.12.9 Nearest integer functions

7.12.9.1 The ceil type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F ceil(R x);
```

Description

2 The **ceil** type-generic macro ~~compute~~ computes the smallest integer value not less than x that is representable in *F*.

Returns

3 The **ceil** type-generic macro ~~return~~ returns $\lceil \mathtt{x} \rceil$, expressed as a floating-point number.

7.12.9.2 The floor type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F floor(R x);
```

Description

2 The **floor** type-generic macro ~~compute~~ computes the largest integer value not greater than x that is representable in *F*.

Returns

3 The **floor** type-generic macro return $\lfloor \mathtt{x} \rfloor$, expressed as a floating-point number.

7.12.9.3 The nearbyint type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F nearbyint(R x);
```

Description

2 The **nearbyint** type-generic macro ~~round their~~ rounds its argument to an integer value in floating-point format, using the current rounding direction and without raising the "inexact" floating-point exception.

Returns

3 The **nearbyint** type-generic macro ~~return~~ returns the rounded integer value.

7.12.9.4 The rint type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double rint(double x);
float rintf(float x);
```

```
long double rintl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F rint(R x);
```

Description

2 The **rint** type-generic macro ~~differ~~ differs from the **nearbyint** ~~functions (??~~macro (7.12.9.3) only in that the ~~**rint** functions~~ **rint** type-generic macro may raise the "inexact" floating-point exception if the result differs in value from the argument.

Returns

3 The **rint** type-generic macro ~~return~~ returns the rounded integer value. ~~Synopsis replace~~

```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

7.12.9.5 The lrint and llrint type-generic macros

Synopsis

1
```
#include <math.h>
constexpr long int lrint(R x);
constexpr long long int llrint(R x);
```

Description

2 The **lrint** and **llrint** ~~functions~~ type-generic macros round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lrint** and **llrint** ~~functions~~ type-generic macros return the rounded integer value.

7.12.9.6 The round type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F round(R x);
```

Description

2 The **round** type-generic macro ~~round their~~ rounds its argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Returns

3 The **round** type-generic macro ~~return~~ returns the rounded integer value.

7.12.9.7 The `lround` and `llround` type-generic macros

~~Synopsis replace~~

```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr long int lround(R x);
constexpr long long int llround(R x);
```

Description

2 The **lround** and **llround** ~~functions~~ type-generic macros round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lround** and **llround** ~~functions~~ type-generic macros return the rounded integer value.

7.12.9.8 The `trunc` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Synopsis

1
```
#include <math.h>
constexpr F trunc(R x);
```

Description

2 The **trunc** type-generic macro ~~round their~~ rounds its argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

3 The **trunc** type-generic macro ~~return~~ returns the truncated integer value.

7.12.10 Remainder functions

7.12.10.1 The `fmod` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F fmod(R x, S y);
```

Description

2 The **`fmod`** type-generic macro ~~compute~~ computes the floating-point remainder of `x`/`y`.

Returns

3 The **`fmod`** type-generic macro ~~return~~ returns the value `x` $-$ n`y`, for some integer n such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`. If `y` is zero, whether a domain error occurs or the **`fmod`** functions return zero is implementation-defined.

7.12.10.2 The `remainder` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F remainder(R x, S y);
```

Description

2 The **`remainder`** type-generic macro ~~compute~~ computes the remainder `x` REM `y` required by IEC 60559.[330)]

Returns

3 The **`remainder`** type-generic macro ~~return~~ returns `x` REM `y`. If `y` is zero, whether a domain error occurs or the functions return zero is implementation-defined.

7.12.10.3 The `remquo` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y,
      int *quo);
```

Synopsis

1
```
#include <math.h>
F remquo(R x, S y, int *quo);
```

Description

2 The **`remquo`** type-generic macro ~~compute~~ computes the same remainder as the **`remainder`** functions. In the object pointed to by `quo` ~~they store~~ it stores a value whose sign is the sign of `x`/`y` and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of `x`/`y`, where n is an implementation-defined integer greater than or equal to 3.

Returns

3 The **`remquo`** type-generic macro ~~return~~ returns `x` REM `y`. If `y` is zero, the value stored in the object pointed to by `quo` is unspecified and whether a domain error occurs or the ~~functions return~~ macro returns zero is implementation defined.

330)"When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of $\frac{x}{y}$; whenever $|n - \frac{x}{y}| = \frac{1}{2}$, then n is even. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

7.12.11 Manipulation functions

7.12.11.1 The copysign type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F copysign(R x, S y);
```

Description

2 The **copysign** type-generic macro ~~produce~~ produces a value with the magnitude of x and the sign of y. ~~They produce~~ It produces a NaN (with the sign of y) if x is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the **copysign** type-generic macro ~~regard~~ regards the sign of zero as positive.

Returns

3 The **copysign** type-generic macro ~~return~~ returns a value with the magnitude of x and the sign of y.

7.12.11.2 The nan functions

Synopsis

1
```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

2 The **nan**, **nanf**, and **nanl** functions convert the string pointed to by tagp according to the following rules. The call **nan**("*n-char-sequence*") is equivalent to **strtod**("NAN(*n-char-sequence*)", ~~(char**)NULL)~~ nullptr); the call **nan**("") is equivalent to ~~**strtod**("NAN()",(char**)NULL)~~ **strtod**("NAN()", **nullptr**). If tagp does not point to an n-char sequence or an empty string, the call is equivalent to ~~**strtod**("NAN",(char**)NULL)~~ **strtod**("NAN", **nullptr**). Calls to **nanf** and **nanl** are equivalent to the corresponding calls to **strtof** and **strtold**.

Returns

3 The **nan** functions return a quiet NaN, if available, with content indicated through tagp. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3).

7.12.11.3 The nextafter type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F nextafter(R x, F y);
```

Description

2 The **nextafter** type-generic macro ~~determine~~ determines the next representable value, in the type of the function, after `x` in the direction of `y`, where `x` and `y` are first converted to ~~the type of the function~~*F*. The type of *F* is only inferred from `x` and not from `y`. The **nextafter** type-generic macro ~~return~~ returns `y` if `x` equals `y`. A range error may occur if the magnitude of `x` is the largest finite value representable in the type and the result is infinite or not representable in the type.

Returns

3 The **nextafter** type-generic macro ~~return~~ returns the next representable value in the specified format after `x` in the direction of `y`. For finite results, the returned value shall be either a zero, a subnormal floating-point number, or a normalized floating-point number.

7.12.11.4 The `nexttoward` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F nexttoward(R x, long double y);
```

Description

2 The **nexttoward** type-generic macro ~~are~~ is equivalent to the **nextafter** ~~functions~~ macro except that the second parameter has type **long double** ~~or _Decimal128 and the functions return~~ and the macro returns `y` converted to ~~the typeof the function if x equals y.~~ *F* if `x` equals `y`.[331)]

7.12.11.5 The `carg` type-generic macro

Synopsis

1
```
#include <math.h>
constexpr F carg(C z);
```

Constraints

2 *C* shall be an arithmetic type.

Description

3 The **carg** type-generic macro computes the argument (also called phase angle) of `z`, with a branch cut along the negative real axis. Return type and value are the same as

```
atan2(imaginary_value(z), real_value(z)).
```

4 Outside a function call, this macro can be converted to function pointer of type `F(*)(C)` where `C` is an arithmetic type and where `F` is the corresponding real type.

Returns

5 The **carg** type-generic macro returns the value of the argument in the interval $[-\pi, +\pi]$.

7.12.11.6 The `conj` type-generic macro

Synopsis

1
```
#include <math.h>
constexpr C conj(C z);
```

[331)] The result of the **nexttoward** type-generic macro is determined in *F*, without loss of range or precision in a floating second argument.

Constraints

2 *C* shall be an arithmetic type.

Description

3 The **conj** type-generic macro computes the complex conjugate of z, by reversing the sign of its imaginary part. Return type and value are the same as

```
real_value(z)+((generic_type(z))(-1.0if × imaginary_value(z)))
```

If *C* is a real type, the return value is z.

4 If z is a constant expression so is **conj**(z), only that if *C* is an integer type, **conj**(z) is not an integer constant expresssion.

5 Outside a function call, this macro can be converted to function pointer of type C(*)(C) where C is an arithmetic type.

Returns

6 The **conj** type-generic macro returns the complex conjugate value in the same type.

7.12.11.7 The cproj type-generic macro

Synopsis

1
```
#include <math.h>
constexpr C cproj(C z);
```

Constraints

2 *C* shall be an arithmetic type.

Description

3 The **cproj** type-generic macro computes a projection of z onto the Riemann sphere: z projects to z except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *C* is a real floating type and z is infinite, then **cproj**(z) is equivalent to **INFINITY**. If *C* is a complex type and z has an infinite part, then **cproj**(z) is equivalent to

```
INFINITY + 1.0if × copysign(0.0f, cimag(z))
```

Otherwise, the return value is z.

4 Outside a function call, this macro can be converted to function pointer of type C(*)(C) where C is an arithmetic type.

Returns

5 The **cproj** type-generic macro returns the value of the projection onto the Riemann sphere.

7.12.12 Maximum, minimum, and positive difference functions

7.12.12.1 The fdim type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F fdim(R x, S y);
```

Description

2 The **fdim** type-generic macro ~~determine~~ determines the *positive difference* between ~~their~~ its arguments:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \le y \end{cases}$$

A range error may occur.

Returns

3 The **fdim** type-generic macro ~~return~~ returns the positive difference value.

7.12.12.2 The fmax type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F fmax(R x, S y);
```

Description

2 The **fmax** type-generic macro ~~determine~~ determines the maximum numeric value of their arguments.[332)]

Returns

3 The **fmax** type-generic macro return the maximum numeric value of their arguments.

7.12.12.3 The fmin type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Synopsis

1
```
#include <math.h>
constexpr F fmin(R x, S y);
```

Description

2 The **fmin** type-generic macro ~~determine~~ determines the minimum numeric value of their arguments.[333)]

Returns

3 The **fmin** type-generic macro ~~return~~ returns the minimum numeric value of their arguments.

7.12.12.4 The math_pdiff type-generic macro

[332)] NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the **fmax** type-generic macro chooses the numeric value. See F.10.9.2.

[333)] The **fmin** type-generic macro are analogous to the **fmax** functions in their treatment of NaNs.

Synopsis

1
```
#include <math.h>
constexpr U math_pdiff(R x, S y);
```

Description

2 The **math_pdiff** type-generic macro determines the *positive difference* between the arguments:

$$\begin{cases} \mathtt{x} - \mathtt{y} & \text{if } \mathtt{x} > \mathtt{y} \\ +0 & \text{if } \mathtt{x} \leq \mathtt{y} \end{cases}$$

3 If any of the arguments has a floating point type, the result value and type is the same as for the **fdim** type-generic macro. Otherwise, if k is the highest rank amoung the promoted types R and S, U is the unsigned integer type with rank k. Then, the mathematical result fits into the target type and no error may occur.

4 Outside a function call, the **math_pdiff** type-generic macro can be converted to a function pointer type `R(*)(R, R)` where `R` where is floating point type, or to `U(*)(R, S)` where `R` and `S` are wide integer types, and where `U` is the corresponding unsigned type with rank as described above.

Returns

5 The **math_pdiff** type-generic macro returns the positive difference value.

7.12.12.5 The max type-generic macro

Synopsis

1
```
#include <math.h>
constexpr Q max(R x, S y);
```

Description

2 The **max** type-generic macro determines the maximum numeric value of its arguments.[334] If any of the arguments has a floating point type, the result value and type is the same as for the **fmax** type-generic macro. Otherwise, the result type Q is the type after usual arithmetic conversions. If one of the types is signed, the other is unsigned and `Q` is also unsigned, instead of being converted modulo a negative argument value is replaced by 0. Then, the mathematical result fits into the target type and no error may occur.

3 Outside a function call, the **max** type-generic macro can be converted to a function pointer type `R(*)(R, R)` where `R` is a floating point type, or to `Q(*)(R, S)` where `R` and `S` are wide integer types, and where `Q` is the type after usual arithmetic conversions for `R` and `S`.

Returns

4 The **max** type-generic macro returns the maximum numeric value of its arguments.

7.12.12.6 The min type-generic macro

Synopsis

1
```
#include <math.h>
constexpr Q min(R x, S y);
```

Description

2 The **min** type-generic macro determines the minimum numeric value of its arguments.[335] If any of the arguments has a floating point type, the result value and type is the same as for the **fmin** type-generic macro. Otherwise, the result type Q is determined as follows. If, after promotion,

[334] NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the **max** type-generic macro chooses the numeric value. See F.10.9.2.

[335] The **min** type-generic macro are analogous to the **max** functions in their treatment of NaNs.

R and *S* are both unsigned types, *Q* is the type of the two with the least integer rank. If, after promotion, *R* and *S* are both signed types *Q* is type of the two with the highest integer rank. Otherwise, *Q* is the signed type of the two. If an argument value is greater than the maximum value for *Q*, instead of beeing converted, it is replaced by that maximum value. Then, the mathematical result fits into the target type and no error may occur.

3 Outside a function call, the **min** type-generic macro can be converted to a function pointer type `R(*)(R, R)` where `R` is a floating point type, or to `Q(*)(R, S)` where `R` and `S` are wide integer types, and where `Q` is the type *Q* as described above.

Returns

4 The **min** type-generic macro returns the minimum numeric value of its arguments.

7.12.13 Floating multiply-add

7.12.13.1 The `fma` type-generic macro

~~Synopsis replace~~

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

Synopsis

1
```
#include <math.h>
constexpr F fma(R x, S y, T z);
```

Description

2 The **fma** type-generic macro ~~compute~~ computes $(\mathtt{x} \times \mathtt{y}) + \mathtt{z}$, rounded as one ternary operation: ~~they compute~~ it computes the value (as if) to infinite precision and ~~round~~ rounds once to the result format, according to the current rounding mode. A range error may occur.

Returns

3 The **fma** type-generic macro ~~return~~ returns $(\mathtt{x} \times \mathtt{y}) + \mathtt{z}$, rounded as one ternary operation.

7.12.14 Comparison macros

1 The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships — *less*, *greater*, and *equal* — is true. Relational operators may raise the "invalid" floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[336] The following subclauses provide macros that are *quiet* (non floating-point exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the "invalid" floating-point exception. In the synopses in this subclause, ~~*real-floating* indicates that the argument shall be an expression of real floating type~~[337] *R* and *S* indicate the real floating argument types.~~(both arguments need not have the same type)~~[337] Both arguments shall be converted according to the usual arithmetic conversions.[338]

2 Outside a function call, these macros can be converted to function pointer types `bool(*)(R, R)` where `R` is a floating point type.

7.12.14.1 The `isgreater` macro

[336] IEC 60559 requires that the built-in relational operators raise the "invalid" floating-point exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

[337] ~~If any argument is of integer type, or any other type that is not a real floating type, the behavior is undefined.~~

[337] If any argument is of integer type, or any other type that is not a real floating type, the behavior is undefined.

[338] Whether an argument represented in a format wider than its semantic type is converted to the semantic type is unspecified.

~~Synopsis replace~~

```
#include <math.h>
int isgreater(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool isgreater(R x, S y);
```

Description

2 The `isgreater` macro determines whether its first argument is greater than its second argument. The value of `isgreater(x,y)` is always equal to `(x)> (y)`; however, unlike `(x)> (y)`, `isgreater(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered.

Returns

3 The `isgreater` macro returns the value of `(x)> (y)`.

7.12.14.2 The `isgreaterequal` macro

~~Synopsis replace~~

```
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool isgreaterequal(R x, S y);
```

Description

2 The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x,y)` is always equal to ~~`(x)>= (y)`~~ (x)≥(y) ; however, unlike ~~`(x)>= (y)`~~ (x)≥(y) , `isgreaterequal(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered.

Returns

3 The `isgreaterequal` macro returns the value of ~~`(x)>= (y)`~~ (x)≥(y) .

7.12.14.3 The `isless` macro

~~Synopsis replace~~

```
#include <math.h>
int isless(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool isless(R x, S y);
```

Description

2 The `isless` macro determines whether its first argument is less than its second argument. The value of `isless(x,y)` is always equal to `(x)< (y)`; however, unlike `(x)< (y)`, `isless(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered.

Returns

3 The `isless` macro returns the value of (x) < (y).

7.12.14.4 The `islessequal` macro

~~Synopsis replace~~

```
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool islessequal(R x, S y);
```

Description

2 The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal**(x,y) is always equal to ~~(x) <= (y)~~ (x) ≤ (y) ; however, unlike ~~(x) <= (y)~~ (x) ≤ (y) , **islessequal**(x,y) does not raise the "invalid" floating-point exception when x and y are unordered.

Returns

3 The **islessequal** macro returns the value of ~~(x) <= (y)~~ (x) ≤ (y) .

7.12.14.5 The islessgreater macro

~~Synopsis replace~~

```
#include <math.h>
int islessgreater(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool islessgreater(R x, S y);
```

Description

2 The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater**(x,y) macro is similar to ~~(x) < (y) || (x) > (y)~~ (x) < (y) ∨ (x) > (y) ; however, **islessgreater**(x,y) does not raise the "invalid" floating-point exception when x and y are unordered (nor does it evaluate x and y twice).

Returns

3 The **islessgreater** macro returns the value of ~~(x) < (y) || (x) > (y)~~ (x) < (y) ∨ (x) > (y) .

7.12.14.6 The isunordered macro

~~Synopsis replace~~

```
#include <math.h>
int isunordered(real-floating x, real-floating y);
```

Synopsis

1
```
#include <math.h>
constexpr bool isunordered(R x, S y);
```

Description

2 The **isunordered** macro determines whether its arguments are unordered.

Returns

3 The **isunordered** macro returns ~~1~~ **true** if its arguments are unordered and ~~0 otherwise.~~ **false** otherwise.

7.12.15 Type properties and values

Synopsis

1
```
#include <math.h>
constexpr bool isice(A x);       // x is an integer constant expression
constexpr bool isvla(A x);       // x is an variably modified array
constexpr bool iscomplex(A x);   // x has complex type
constexpr bool isconstant(A x); // x is a scalar constant expression
constexpr bool isextended(A x); // x has an extended integer type
constexpr bool isfloating(A x); // x has floating type
constexpr bool isfunction(A x); // x has function or function pointer type
constexpr bool isinteger(A x);  // x has integer type
constexpr bool isnarrow(A x);   // x has a narrow integer type
constexpr bool isnull(A x);     // x is integer and a null pointer constant
constexpr bool ispointer(A x);  // x has pointer, array or function type
constexpr bool isreal(A x);     // x has real type
constexpr bool issigned(A x);   // x has signed type
constexpr bool isunsigned(A x); // x has unsigned type
constexpr bool iswide(A x);     // x has a wide integer type
constexpr A tohighest(A x);     // maximum of the type of x
constexpr A tolowest(A x);      // minimum of the type of x
constexpr A toone(A x);         // the difference from 1 to the smallest
                                // value strictly greater than 1
constexpr A tozero(A x);        // the difference from 0 to the
                                // smallest value strictly greater than 0
```

Description

2 The macros query properties of arithmetic or scalar types or values as indicated in the comments. Their argument is not evaluated and only serves for the type information. For the macros starting with **is**, the result is an integer constant expression of type **bool**; for the macros starting with **to** the result is an arithmetic constant expression of the same type as their argument, and an integer constant expression if the type is an integer type.

3 **NOTE** The following shows implementations of some of these macros in terms of other features that are provided in clause 6.

```
#define isinteger(X)                \
    generic_selection((X)+0ULL,     \
        unsigned long long: true,   \
        default: false)

#define isunsigned(X) (isinteger(X) ∧ ((real_type(X))-1 < 0))
#define issigned(X)   (isinteger(X) ∧ ((real_type(X))-1 > 0))

#define isfloating(X)               \
    generic_selection(real_type(X), \
        float: true,                \
        double: true,               \
        long double: true,          \
        default: false)

#define tohighest(X)                                                    \
    generic_selection((X),                                              \
        char:        (char)(((unsigned char)-1)/(isunsigned(X) ? 1 : 2)),   \
        signed char:               (signed char)(((unsigned char)-1)/2),   \
        signed short:             (signed short)(((unsigned short)-1)/2),  \
        signed int:                  (signed int)(((unsigned int)-1)/2),   \
        signed long:               (signed long)(((unsigned long)-1)/2),   \
        signed long long:    (signed short)(((unsigned long long)-1)/2),   \
        float:                                                HUGE_VALF,   \
        double:                                                HUGE_VAL,   \
        long double:                                          HUGE_VALL,   \
```

```
          default: ((generic_type(X))-1))


#define tolowest(X) ((generic_type(X))(isfloating(X) ? -tohighest(X) : -tohighest(X)-1))

#define isice(X) generic_selection(tonullptr((X)-(X)), nullptr_t: true, default: false)
#define isvla(X) (¬isice(sizeof(X)))
```

7.13 Nonlocal jumps <setjmp.h>

1 The header `<setjmp.h>` defines the macro **setjmp**, and declares one function and one type, for bypassing the normal function call and return discipline.[339)]

2 The type declared is

```
jmp_buf
```

which is ~~an~~ a complete opaque array type suitable for holding the information needed to restore a calling environment. The environment of a call to the **setjmp** macro consists of information sufficient for a call to the **longjmp** function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

3 It is unspecified whether **setjmp** is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name **setjmp**, the behavior is undefined.

7.13.1 Save calling environment

7.13.1.1 The setjmp macro

Synopsis

1
```
#include <setjmp.h>
int setjmp(jmp_buf env);
int setjmp([[core::writethrough]] jmp_buf env);
```

Description

2 The **setjmp** macro saves its calling environment in its **jmp_buf** argument for later use by the **longjmp** function.

Returns

3 If the return is from a direct invocation, the **setjmp** macro returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** macro returns a nonzero value.

Environmental limits

4 An invocation of the **setjmp** macro shall appear only in one of the following contexts:

— the entire controlling expression of a selection or iteration statement;

— one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;

— the operand of a unary ~~!~~ ¬ operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or

— the entire expression of an expression statement (possibly cast to **void**).

5 If the invocation appears in any other context, the behavior is undefined.

7.13.2 Restore calling environment

7.13.2.1 The longjmp function

Synopsis

1
```
#include <setjmp.h>
_Noreturn void longjmp(jmp_buf env, int val);
```

[339)]These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

Description

2 The `longjmp` function restores the environment saved by the most recent invocation of the `setjmp` macro in the same invocation of the program with the corresponding `jmp_buf` argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the `setjmp` macro has terminated execution[340] in the interim, or if the invocation of the `setjmp` macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine[341] have state, as of the time the `longjmp` function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding `setjmp` macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call are indeterminate.

Returns

4 After `longjmp` is completed, thread execution continues as if the corresponding invocation of the `setjmp` macro had just returned the value specified by `val`. The `longjmp` function cannot cause the `setjmp` macro to return the value 0; if `val` is 0, the `setjmp` macro returns the value 1.

5 **EXAMPLE** The `longjmp` function that returns control back to the point of the `setjmp` invocation might cause ~~memory~~ the storage instance associated with a variable length array object to be squandered.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
      int x[n];          // valid:  f is not terminated
      setjmp(buf);
      g(n);
}

void g(int n)
{
      int a[n];          // a may remain allocated
      h(n);
}

void h(int n)
{
      int b[n];          // b may remain allocated
      longjmp(buf, 2);   // might cause memory loss
}
```

340) For example, by executing a `return` statement or because another `longjmp` call has caused a transfer to a `setjmp` invocation in a function earlier in the set of nested calls.

341) This includes, but is not limited to, the floating-point status flags and the state of open files.

7.14 Signal handling `<signal.h>`

1 The header `<signal.h>` declares ~~a type~~ two types and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

2 The types defined are

```
sig_atomic_t
```

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts; and

```
typedef void (*sighandler_t)(int);
```

3 The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the **signal** function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type **int** and distinct values that are the signal numbers, each corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the **abort** function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

4 An implementation need not generate any of these signals, except as a result of explicit calls to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter,[342] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

7.14.1 Specify signal handling

7.14.1.1 The signal function

Synopsis

1
```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
sighandler_t signal(int signum, sighandler_t handler) [[core::modifies(errno)]];
```

Description

2 The **signal** function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is **SIG_DFL**, default handling for that signal will occur. If the value of `func` is **SIG_IGN**, the signal will be ignored. Otherwise, `func` shall point to a

[342] See "future library directions" (7.32.7). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library),[343] is called a *signal handler*.

3 When a signal occurs and `func` points to a function, it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of `SIGILL`, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

4 If the signal occurs as the result of calling the `abort` or `raise` function, the signal handler shall not call the `raise` function.

5 If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or the signal handler calls any function in the standard library other than

— the `abort` function,

— the `_Exit` function,

— the `quick_exit` function,

— the functions and generic functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,

— the `atomic_is_lock_free` generic function with any atomic argument, or

— the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the `signal` function results in a `SIG_ERR` return, the value of `errno` is indeterminate.[344]

6 At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

7 Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the `signal` function.

Returns

8 If the request can be honored, the `signal` function returns the value of `func` for the most recent successful call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

Recommended practice

9 It is recommended that implementations apply the `core::reentrant` attribute to C library functions where they make such guarantees. Whenever possible, implementations should issue

[343]This includes functions called indirectly via standard library functions (e.g., a `SIGABRT` handler called via the `abort` function).

[344]If any signal is generated by an asynchronous signal handler, the behavior is undefined.

a diagnostic if a signal handler is used as an argument to **signal** function that has not the **core::reentrant** attribute.

Forward references: the **abort** function (7.22.5.1), the **exit** function (7.22.5.4), the **_Exit** function (7.22.5.5), the **quick_exit** function (7.22.5.7).

7.14.2 Send signal

7.14.2.1 The raise function

Synopsis

1
```
#include <signal.h>
int raise(int sig);
```

Description

2 The **raise** function carries out the actions described in 7.14.1.1 for the signal sig. If a signal handler is called, the **raise** function shall not return until after the signal handler does.

Returns

3 The **raise** function returns zero if successful, nonzero if unsuccessful.

7.15 Alignment `<stdalign.h>`

~~The header defines four macros.~~

1 The obsolescent header `<stdalign.h>` defines two macros that are suitable for use in **`#if`** preprocessing directives. They are

```
__alignas_is_defined
```

and

```
__alignof_is_defined
```

which both expand to the integer constant 1.

7.16 Variable arguments <stdarg.h>

1 The header <stdarg.h> declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

2 A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

3 The type declared is

```
va_list
```

which is a complete opaque object type suitable for holding information needed by the macros **va_start**, **va_arg**, **va_end**, and **va_copy**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type **va_list**. The object ap may be passed as an argument to another function; if that function invokes the **va_arg** macro with parameter ap, the value of ap in the calling function is indeterminate and shall be passed to the **va_end** macro prior to any further reference to ap.[345)]

7.16.1 Variable argument list access macros

1 The **va_start** and **va_arg** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **va_copy** and **va_end** are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the **va_start** and **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro in the same function.

7.16.1.1 The va_arg macro

Synopsis

1
```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

2 The **va_arg** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter ap shall have been initialized by the **va_start** or **va_copy** macro (without an intervening invocation of the **va_end** macro for the same ap). Each invocation of the **va_arg** macro modifies ap so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

— one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;

— one type is pointer to **void** and the other is a pointer to a character type.

Returns

3 The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

[345)]It is permitted to create a pointer to a **va_list** and pass that pointer to another function, in which case the original function can make further use of the original list after the other function returns.

7.16.1.2 The `va_copy` macro

Synopsis

1
```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
void va_copy( [[ core::writethrough ]] va_list dest, va_list src);
```

Description

2 The `va_copy` macro initializes `dest` as a copy of `src`, as if the `va_start` macro had been applied to `dest` followed by the same sequence of uses of the `va_arg` macro as had previously been used to reach the present state of `src`. Neither the `va_copy` nor `va_start` macro shall be invoked to reinitialize `dest` without an intervening invocation of the `va_end` macro for the same `dest`.

Returns

3 The `va_copy` macro returns no value.

7.16.1.3 The `va_end` macro

Synopsis

1
```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

2 The `va_end` macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the `va_start` macro, or the function containing the expansion of the `va_copy` macro, that initialized the `va_list` `ap`. The `va_end` macro may modify `ap` so that it is no longer usable (without being reinitialized by the `va_start` or `va_copy` macro). If there is no corresponding invocation of the `va_start` or `va_copy` macro, or if the `va_end` macro is not invoked before the return, the behavior is undefined.

Returns

3 The `va_end` macro returns no value.

7.16.1.4 The `va_start` macro

Synopsis

1
```
#include <stdarg.h>
void va_start(va_list ap, parmN);
void va_start( [[ core::writethrough ]] va_list ap, parmN);
```

Description

2 The `va_start` macro shall be invoked before any access to the unnamed arguments.

3 The `va_start` macro initializes `ap` for subsequent use by the `va_arg` and `va_end` macros. Neither the `va_start` nor `va_copy` macro shall be invoked to reinitialize `ap` without an intervening invocation of the `va_end` macro for the same `ap`.

4 The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the ~~`, ...`~~ `, ...`). If the parameter *parmN* is declared with ~~the `register` storage class, with~~ a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

5 The `va_start` macro returns no value.

6 **EXAMPLE 1** The function `f1` gathers into an array a list of arguments that are pointers to strings (but not more than `MAXARGS` arguments), then passes the array as a single argument to function `f2`. The number of pointers is specified by the first argument to `f1`.

```
#include <stdarg.h>
#define MAXARGS   31

void f1(int n_ptrs, ...)
void f1(int n_ptrs, ...)
{
        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;

        if (n_ptrs > MAXARGS)
                n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no  <  n_ptrs)
                array[ptr_no++] = va_arg(ap, char *);
        va_end(ap);
        f2(n_ptrs, array);
}
```

Each call to `f1` is required to have visible the definition of the function or a declaration such as

```
void f1(int, ...);
void f1(int, ...);
```

7 **EXAMPLE 2** The function `f3` is similar, but saves the status of the variable argument list after the indicated number of arguments; after `f2` has been called once with the whole list, the trailing part of the list is gathered again and passed to function `f4`.

```
#include <stdarg.h>
#define MAXARGS 31

void f3(int n_ptrs, int f4_after, ...)
void f3(int n_ptrs, int f4_after, ...)
{
        va_list ap, ap_save;
        char *array[MAXARGS];
        int ptr_no = 0;
        if (n_ptrs > MAXARGS)
                n_ptrs = MAXARGS;
        va_start(ap, f4_after);
        while (ptr_no < n_ptrs) {
                array[ptr_no++] = va_arg(ap, char *);
                if (ptr_no == f4_after)
                if (ptr_no ≡ f4_after)
                        va_copy(ap_save, ap);
        }
        va_end(ap);
        f2(n_ptrs, array);

        // Now process the saved copy.

        n_ptrs -= f4_after;
        ptr_no = 0;
        while (ptr_no < n_ptrs)
                array[ptr_no++] = va_arg(ap_save, char *);
        va_end(ap_save);
        f4(n_ptrs, array);
}
```

7.17 Atomics `<stdatomic.h>`

7.17.1 Introduction

1 The header `<stdatomic.h>` defines several macros and declares several types and functions for performing atomic operations on data shared between threads.[346]

2 Implementations that define the macro `__CORE_NO_ATOMICS__` need not provide this header nor support any of its facilities.

3 The macros defined are

```
__CORE_VERSION_STDATOMIC_H__
```

which expands to `202005L`[347] and the *atomic lock-free macros*

```
ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
```

which expand to constant expressions suitable for use in **`#if`** preprocessing directives and which indicate the lock-free property of the corresponding atomic types (both signed and unsigned).

4 The types include

```
memory_order
```

which is an enumerated type whose enumerators identify memory ordering constraints;

```
atomic_flag
```

which is a complete opaque object type representing a lock-free, primitive atomic flag; and several atomic analogs of integer types.

5 In the following synopses:

— An *A* refers to an atomic type that is not **`const`**-qualified.

— A *C* refers to its ~~corresponding non-atomic~~ generic type.

— An *M* refers to the type of the other argument for arithmetic operations. For atomic ~~integer~~ arithmetic types, *M* is *C*. For atomic pointer types, *M* is **`ptrdiff_t`**.

— The functions not ending in **`_explicit`** have the same semantics as the corresponding **`_explicit`** function with **`memory_order_seq_cst`** for the **`memory_order`** argument.

6 ~~It is unspecified whether any generic function declared in is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is undefined.~~

[346] See "future library directions" (7.32.8).

[347] The intent of this macro is to keep track of the version of this document to which a particular C library implementation of `<stdatomic.h>` adheres. This is meant to facilitate the transition to a new standard for users, not as a leeway for implementations to delay an upgrade.

~~Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects.~~ The prototype for a call to a type-generic macro specified in this clause is determined by the pointed-to type *A* of the first argument of the call and the types *C* and *M* are deduced according to the above rules. Other arguments to the call shall be implicitly convertible to the types that are required by the chosen prototype and are converted accordingly before the call.

7 ~~Synopsis replace~~ When not used within a function call, any type-generic macro specified in this clause may be converted to a function pointer with the same prototype, where *A* is any non-const atomic type and *C* (and *M* if necessary) are derived as above.

~~**Description**~~

7.17.2 Initialization

1 ~~The expands to a token sequence suitable for initializing an atomic object of a type that is initialization-compatible with `value`.~~ An atomic object with automatic storage duration that is not explicitly initialized is initially in an indeterminate state; however, the default (zero) initialization for objects with static or thread-local storage duration is guaranteed to produce a valid state.

2 Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

3 **EXAMPLE 1** The following definitions ensure valid states for `guide` and `head` regardless if these are found in file scope or block scope. Thus any atomic operation that is performed on them after their initialization has been met is well defined.

```
atomic_int guide = ATOMIC_VAR_INIT(42);
atomic_type(int) guide = 42;
static atomic_type(void*) head;
```

4 **EXAMPLE 2** With the following definition in block scope, concurrent accesses to `cumul` are undefined unless a prior race-free initialization, either by a call to **`atomic_init`**, a store operation or by assignment, has been performed.

```
atomic_type(double) cumul;
```

7.17.2.1 The `atomic_init` type-generic macro

Synopsis

1
```
#include <stdatomic.h>
void atomic_init(volatile A *obj, C value);
void atomic_init(A *obj, C value);
```

Description

2 The **`atomic_init`** type-generic macro initializes the atomic object pointed to by `obj` to the value `value`, while also initializing any additional state that the implementation might need to carry for the atomic object.

3 Although this function initializes an atomic object, it does not avoid data races; concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

4 If a signal occurs other than as the result of calling the **`abort`** or **`raise`** functions, the behavior is undefined if the signal handler calls the **`atomic_init`** generic function.

Returns

5 The **`atomic_init`** type-generic macro returns no value.

6 **EXAMPLE**

```
atomic_int guide;
atomic_type(int) guide;
```

```
atomic_init(&guide, 42);
```

7.17.3 Order and consistency

1 The enumerated type `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations as defined in 5.1.2.4 and may provide for operation ordering. Its enumeration constants are as follows:[348)]

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

2 For `memory_order_relaxed`, no operation orders memory.

3 For `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`, a store operation performs a release operation on the affected memory location.

4 For `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`, a load operation performs an acquire operation on the affected memory location.

5 For `memory_order_consume`, a load operation performs a consume operation on the affected memory location.

6 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

— the result of the last modification A of M that precedes B in S, if it exists, or

— if A exists, the result of some modification of M that is not `memory_order_seq_cst` and that does not happen before A, or

— if A does not exist, the result of some modification of M that is not `memory_order_seq_cst`.

7 **NOTE 1** Although it is not explicitly required that S include lock operations, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering.

8 **NOTE 2** Atomic operations specifying `memory_order_relaxed` are relaxed only with respect to memory ordering. Implementations still guarantee that any given atomic access to a particular atomic object is indivisible with respect to all other atomic accesses to that object.

9 For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

10 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

11 For atomic modifications A and B of an atomic object M, B occurs later than A in the modification order of M if:

— there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, or

— there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, or

[348)]See "future library directions" (7.32.8).

— there are **memory_order_seq_cst** fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

12 **NOTE 3** The memory orderings of **memory_order** impose different ordering constraints on certain operations. **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_acq_rel** and **memory_order_seq_cst** form an inclusive chain of such constraints, from weakest to strongest. **memory_order_release** imposes constraints that are incompatible with **memory_order_consume** and **memory_order_acquire**, and that are stronger than **memory_order_relaxed** and weaker than **memory_order_acq_rel**.

13 Atomic read-modify-write operations shall always read the last value (in the modification order) stored before the write associated with the read-modify-write operation.

14 An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that

— If an evaluation B observes a value computed by A in a different thread, then B does not happen before A.

— If an evaluation A is included in the sequence, then all evaluations that assign to the same variable and happen before A are also included.

15 **NOTE 4** The second requirement disallows "out-of-thin-air", or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations can appear in this sequence out of thread order. For example, with x and y initially zero,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, 42, memory_order_relaxed);
```

is allowed to produce ~~r1 == 42 && r2 == 42~~ r1 $\equiv$ 42 $\wedge$ r2 $\equiv$ 42. The sequence of evaluations justifying this consists of:

```
atomic_store_explicit(&y, 42, memory_order_relaxed);
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);
r2 = atomic_load_explicit(&x, memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, r2, memory_order_relaxed);
```

is not allowed to produce ~~r1 == 42 && r2 == 42~~ r1 $\equiv$ 42 $\wedge$ r2 $\equiv$ 42, since there is no sequence of evaluations that results in the computation of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than **memory_order_acq_rel** ordering, the second requirement has no impact.

Recommended practice

16 The requirements do not forbid ~~r1 == 42 && r2 == 42~~ r1 $\equiv$ 42 $\wedge$ r2 $\equiv$ 42 in the following example, with x and y initially zero:

```
// Thread 1:
r1 = atomic_load_explicit(&x, memory_order_relaxed);
~~if (r1 == 42)~~
```

```
    if (r1 ≡ 42)
        atomic_store_explicit(&y, r1, memory_order_relaxed);

    // Thread 2:
    r2 = atomic_load_explicit(&y, memory_order_relaxed);
    if (r2 == 42)
    if (r2 ≡ 42)
        atomic_store_explicit(&x, 42, memory_order_relaxed);
```

However, this is not useful behavior, and implementations should not allow it.

17 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

7.17.3.1 The **kill_dependency** macro

Synopsis

1
```
#include <stdatomic.h>
type kill_dependency(type y);
```

Description

2 The **kill_dependency** macro terminates a dependency chain; the argument does not carry a dependency to the return value.

Returns

3 The **kill_dependency** macro returns the value of `y`.

7.17.4 Fences

1 This subclause introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*; a fence with release semantics is called a *release fence*.

2 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

3 A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

4 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

7.17.4.1 The **atomic_thread_fence** function

Synopsis

1
```
#include <stdatomic.h>
void atomic_thread_fence(memory_order order);
```

Description

2 Depending on the value of `order`, this operation:

— has no effects, if ~~`order == memory_order_relaxed`~~ `order ≡ memory_order_relaxed`;

— is an acquire fence, if ~~`order == memory_order_acquire` or `order == memory_order_consume`~~ `order ≡ memory_order_acquire` or `order ≡ memory_order_consume`;

— is a release fence, if ~~`order == memory_order_release`~~ `order` ≡ `memory_order_release` ;

— is both an acquire fence and a release fence, if ~~`order == memory_order_acq_rel`~~ `order` ≡ `memory_order_acq_rel` ;

— is a sequentially consistent acquire and release fence, if ~~`order == memory_order_seq_cst`~~ `order` ≡ `memory_order_seq_cst` .

Returns

3 The **`atomic_thread_fence`** function returns no value.

7.17.4.2 The `atomic_signal_fence` function

Synopsis

1
```
#include <stdatomic.h>
void atomic_signal_fence(memory_order order);
```

Description

2 Equivalent to **`atomic_thread_fence`**`(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

3 **NOTE 1** The **`atomic_signal_fence`** function can be used to specify the order in which actions performed by the thread become visible to the signal handler.

4 **NOTE 2** Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with **`atomic_thread_fence`**, but the hardware fence instructions that **`atomic_thread_fence`** would have inserted are not emitted.

Returns

5 The **`atomic_signal_fence`** function returns no value.

7.17.5 Lock-free property

1 The atomic lock-free macros indicate the lock-free property of ~~integer and address atomic~~ atomic integer and pointer types. A value of 0 indicates that the type is never lock-free; a value of 1 indicates that the type is sometimes lock-free; a value of 2 indicates that the type is always lock-free.

2 **NOTE 1** In addition to the synchronization properties between threads, the lock-free property of a type warrants that operations are perceived indivisible in the presence of interrupts, see 5.1.2.3.

Recommended practice

3 Operations that are lock-free should also be *address-free*. That is, atomic operations on the same memory location via two different addresses will ~~communicate atomically~~synchronize. The implementation should not depend on any ~~per-process~~ execution dependent state. This restriction enables ~~communication via memory mapped into a process~~ synchronization via memory that is mapped into an execution more than once and memory shared between ~~two processes~~concurrent program executions.

7.17.5.1 The `atomic_is_lock_free` type-generic macro

Synopsis

1
```
#include <stdatomic.h>
_Bool atomic_is_lock_free(const volatile A *obj);
bool atomic_is_lock_free(const A *obj) [[core::reentrant]];
```

Description

2 The **`atomic_is_lock_free`** type-generic macro indicates whether or not atomic operations on objects of the type pointed to by `obj` are lock-free.

Returns

3 The **`atomic_is_lock_free`** type-generic macro returns ~~nonzero (true)~~ **`true`** if and only if atomic operations on objects of the type pointed to by the argument are lock-free. In any given program

execution, the result of the lock-free query shall be consistent for all pointers of the same type.[349]

7.17.6 Atomic integer types

1 ~~For~~ If the non-atomic version of the direct type exists, for each line in the following table~~,~~[350] the atomic type name is declared as a type that has the same representation and alignment requirements as the ~~corresponding~~ direct type.[351]

Atomic type name	Direct type
atomic_bool	~~_Atomic _Bool~~ atomic_type(bool)
atomic_char	~~_Atomic char~~ atomic_type(char)
atomic_schar	~~_Atomic signed char~~ atomic_type(signed char)
atomic_uchar	~~_Atomic unsigned char~~ atomic_type(unsigned char)
atomic_short	~~_Atomic short~~ atomic_type(short)
atomic_ushort	~~_Atomic unsigned short~~ atomic_type(unsigned short)
atomic_int	~~_Atomic int~~ atomic_type(int)
atomic_uint	~~_Atomic unsigned int~~ atomic_type(unsigned int)
atomic_long	~~_Atomic long~~ atomic_type(long)
atomic_ulong	~~_Atomic unsigned long~~ atomic_type(unsigned long)
atomic_llong	~~_Atomic long long~~ atomic_type(long long)
atomic_ullong	~~_Atomic unsigned long long~~ atomic_type(unsigned long long)
atomic_char16_t	~~_Atomic char16_t~~ atomic_type(char16_t)
atomic_char32_t	~~_Atomic char32_t~~ atomic_type(char32_t)
atomic_wchar_t	~~_Atomic wchar_t~~ atomic_type(wchar_t)
atomic_int_least8_t	~~_Atomic int_least8_t~~ atomic_type(int_least8_t)
atomic_uint_least8_t	~~_Atomic uint_least8_t~~ atomic_type(uint_least8_t)
atomic_int_least16_t	~~_Atomic int_least16_t~~ atomic_type(int_least16_t)
atomic_uint_least16_t	~~_Atomic uint_least16_t~~ atomic_type(uint_least16_t)
atomic_int_least32_t	~~_Atomic int_least32_t~~ atomic_type(int_least32_t)
atomic_uint_least32_t	~~_Atomic uint_least32_t~~ atomic_type(uint_least32_t)
atomic_int_least64_t	~~_Atomic int_least64_t~~ atomic_type(int_least64_t)
atomic_uint_least64_t	~~_Atomic uint_least64_t~~ atomic_type(uint_least64_t)
atomic_int_fast8_t	~~_Atomic int_fast8_t~~ atomic_type(int_fast8_t)
atomic_uint_fast8_t	~~_Atomic uint_fast8_t~~ atomic_type(uint_fast8_t)
atomic_int_fast16_t	~~_Atomic int_fast16_t~~ atomic_type(int_fast16_t)
atomic_uint_fast16_t	~~_Atomic uint_fast16_t~~ atomic_type(uint_fast16_t)
atomic_int_fast32_t	~~_Atomic int_fast32_t~~ atomic_type(int_fast32_t)
atomic_uint_fast32_t	~~_Atomic uint_fast32_t~~ atomic_type(uint_fast32_t)
atomic_int_fast64_t	~~_Atomic int_fast64_t~~ atomic_type(int_fast64_t)
atomic_uint_fast64_t	~~_Atomic uint_fast64_t~~ atomic_type(uint_fast64_t)
atomic_intptr_t	~~_Atomic intptr_t~~ atomic_type(intptr_t)
atomic_uintptr_t	~~_Atomic uintptr_t~~ atomic_type(uintptr_t)
atomic_size_t	~~_Atomic size_t~~ atomic_type(size_t)
atomic_ptrdiff_t	~~_Atomic ptrdiff_t~~ atomic_type(ptrdiff_t)
atomic_intmax_t	~~_Atomic intmax_t~~ atomic_type(intmax_t)
atomic_uintmax_t	~~_Atomic uintmax_t~~ atomic_type(uintmax_t)

Recommended practice

2 The ~~representation~~ alignment of an atomic integer type is not required to ~~have the same size as the corresponding regular~~ be the same as for the non-atomic version of the direct type but it should ~~have the same size~~ be the same whenever possible, as it eases effort required to port existing code. It is recommended that the atomic type name defines exactly the corresponding direct type.

[349] obj can be a null pointer.

[350] See "future library directions" (7.32.8).

[351] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

7.17.7 Operations on atomic types

1 ~~There are only a few kinds of~~ In addition to the operations on atomic ~~types, though there are many instances of those kinds~~ objects that are described by operators, there are a few kinds of operations that are specified as generic functions. This subclause specifies each ~~general kind~~generic function. After evaluation of its arguments, each of these generic functions forms a single read, write or read-modify-write operation with same general properties as described in 5.1.2.4 and 6.2.6.1.

7.17.7.1 The `atomic_store` generic functions

Synopsis

1
```
        #include <stdatomic.h>
        void atomic_store(volatile A *object, C desired);
        void atomic_store_explicit(volatile A *object, C desired, memory_order order);
        void atomic_store(A *object, C desired);
        void atomic_store_explicit(A *object, C desired, memory_order order);
```

Description

2 The `order` argument shall not be **`memory_order_acquire`**, **`memory_order_consume`**, nor **`memory_order_acq_rel`**. Atomically replace the value pointed to by `object` with the value of `desired`. Memory is affected according to the value of `order`.

Returns

3 The **`atomic_store`** generic functions return no value.

7.17.7.2 The `atomic_load` generic functions

Synopsis

1
```
        #include <stdatomic.h>
        C atomic_load(const volatile A *object);
        C atomic_load_explicit(const volatile A *object, memory_order order);
        C atomic_load(const A *object);
        C atomic_load_explicit(const A *object, memory_order order);
```

Description

2 The `order` argument shall not be **`memory_order_release`** nor **`memory_order_acq_rel`**. Memory is affected according to the value of `order`.

Returns

3 Atomically returns the value pointed to by `object`.

7.17.7.3 The `atomic_exchange` generic functions

Synopsis

1
```
        #include <stdatomic.h>
        C atomic_exchange(volatile A *object, C desired);
        C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
        C atomic_exchange(A *object, C desired);
        C atomic_exchange_explicit(A *object, C desired, memory_order order);
```

Description

2 Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (5.1.2.4).

Returns

3 Atomically returns the value pointed to by `object` immediately before the effects.

7.17.7.4 The `atomic_compare_exchange` generic functions

Synopsis

1
```
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected,
        C desired, memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected,
        C desired, memory_order success, memory_order failure);
bool atomic_compare_exchange_strong(A *object, C *expected, C desired);
bool atomic_compare_exchange_strong_explicit(A *object,
          C *expected, C desired, memory_order success, memory_order failure);
bool atomic_compare_exchange_weak(A *object, C *expected, C desired);
bool atomic_compare_exchange_weak_explicit(A *object,
           C *expected, C desired, memory_order success, memory_order failure);
```

Description

2 The `failure` argument shall not be **`memory_order_release`** nor **`memory_order_acq_rel`**. The `failure` argument shall ~~be no stronger than the success~~ not impose more constraints on the operation than the `success` argument.

3 Atomically, compares the contents of the memory pointed to by `object` for equality with that pointed to by `expected`, and if true, replaces the contents of the memory pointed to by `object` with `desired`, and if false, updates the contents of the memory pointed to by `expected` with that pointed to by `object`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. These operations are atomic read-modify-write operations (5.1.2.4).

4 **NOTE 1** For example, the effect of **`atomic_compare_exchange_strong`** is

```
if (memcmp(object, expected, sizeof (*object)) == 0)
if (memcmp(object, expected, sizeof (*object)) ≡ 0)
      memcpy(object, &desired, sizeof (*object));
else
      memcpy(expected, object, sizeof (*object));
```

5 A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return ~~zero~~ **`false`** and store back to `expected` the same memory contents that were originally there.

6 **NOTE 2** This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.

7 **EXAMPLE** A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```
exp = atomic_load(&cur);
do {
      des = function(exp);
} while (!atomic_compare_exchange_weak(&cur, &exp, des));
} while (¬atomic_compare_exchange_weak(&cur, &exp, des));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

Returns

8 The result of the comparison.

7.17.7.5 The atomic fetch and modify generic functions

1 The following operations perform arithmetic and bitwise computations. ~~All of these~~ These operations are applicable to an ~~object of any atomic integer type . None of these operations is applicable to **atomic_bool**.~~ atomic object as long as the non-atomic type can be the left operand

of the corresponding op= compound assignment.[352] The key, operator, and computation correspondence is:

	key	op	computation
	add	+	addition
	sub	-	subtraction
	mult	×	multiplication
	div	/	division
~~lllkey~~	or	~~|~~∪	bitwise inclusive or
	xor	^	bitwise exclusive or
	and	~~&~~∩	bitwise and
	lshift	⧀⊠	left shift
	rshift	⊠⧁	right shift

Synopsis

2

```
#include <stdatomic.h>
~~C atomic_fetch_key(volatile A *object, M operand);~~
~~C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);~~
C atomic_fetch_key(A *object, M operand);
C atomic_fetch_key_explicit(A *object, M operand, memory_order order);
C atomic_key_fetch(A *object, M operand);
C atomic_key_fetch_explicit(A *object, M operand, memory_order order);
```

Description

3 Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4). ~~For signed integer types, arithmetic is defined to use silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.~~

Returns

4 Atomically, the value pointed to by `object` immediately before the effects ~~. The~~ (for **atomic_fetch_***key* variants) or after the effects (for **atomic_***key***_fetch** variants).

5 **NOTE 1** For many aspects the operation of the atomic fetch and modify generic functions are ~~nearly~~ equivalent to the operation of the corresponding *op*= compound assignment operators ~~. The only differences arethat the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated~~ (6.5.17.2). Notable differences are: according to the variant the value returned by the atomic fetch and modify generic functions is the previous value of the ~~object, whereas the value returned by~~ atomic object; the memory order can be specified to be less strict than the operator; the possible range of values for `operand` can be different than for the operator.

6 **NOTE 2** For integer and pointer types, the ~~is the previous value of~~ explicit forms of the ~~atomic object~~atomic fetch and modify generic functions are similar to the following definitions:

```
C atomic_fetch_key_explicit(A *object, M operand, memory_order order){
    C old = atomic_load_explicit(object, memory_order_relaxed);
    while (!atomic_compare_exchange_weak_explicit(
        object, &old,
        old op operand,
        order, memory_order_relaxed));
    return old;
}
C atomic_key_fetch_explicit(A *object, M operand, memory_order order){
    C old = atomic_load_explicit(object, memory_order_relaxed);
    C new;
```

[352]Thus bitwise operations are not permitted for atomic floating point types, and only "add" and "sub" variants are permitted for atomic pointer types. For the latter the type for *M* is **ptrdiff_t**, see 7.17.1.

```
        do {
            new = old op operand;
        } while (!atomic_compare_exchange_weak_explicit(
            object, &old, new, order, memory_order_relaxed));
        return new;
    }
```

As described in 6.5.17.2, the operation "`old` *op* `operand`" will wrap in case of overflow, will never raise a signal or perform a trap, and an unspecified value will be stored for an erroneous operation.

7 **NOTE 3** The non-explicit forms of the atomic fetch and modify generic functions are the same only that `order` is omitted from the declaration and is replaced by **`memory_order_seq_cst`** in the compare-and-exchange operation. As a consequence only the last, successful, compare-and-exchange operation has **`memory_order_seq_cst`** memory order and is the only atomic operation that is visible in the total order of such **`memory_order_seq_cst`** operations.

8 **NOTE 4** For floating-point types, exceptional conditions or floating-point exceptions are handled analogous as described in 6.5.17.2.

9 **EXAMPLE** Provided that the implementation allows such large array sizes, the following use of the `+=` operator is valid. In contrast to that, in preparation of the arguments to the generic function call, a conversion of `large` to **`ptrdiff_t`** has to be performed. This may trap before the call or may give an implementation defined result that differs from the operator version.

```
static const uintmax_t large = 1 + (uintmax_t)PTRDIFF_MAX;
static unsigned char block[large];
static atomic_type(unsigned char*) cp = block;
cp += large;                  // valid, equivalent to cp = &block[large]
atomic_fetch_add(&cp, large); // invalid
```

7.17.8 Atomic flag type and operations

1 The **`atomic_flag`** type provides the ~~classic~~ test-and-set functionality. It has two states, set and clear. In the following synopsis *A* denotes **`atomic_flag`**, possibly **`volatile`** qualified.

2 Operations on an object of type **`atomic_flag`** shall be lock free.~~Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the **`atomic_flag`** type is the minimum hardware-implemented type needed to conform to this document. The remaining types can be emulated with **`atomic_flag`**, though with less than ideal properties.~~[353)]

3 ~~The macro may be used to initialize~~ Implicit or explicit initialization initializes an **`atomic_flag`** to the clear state. An **`atomic_flag`** that is not ~~explicitly initialized with is initially~~ initialized is in an indeterminate state. Concurrent calls (i.e calls that are not related by "happened before") to **`atomic_flag_test_and_set`** or **`atomic_flag_clear`** using such an uninitialized object constitute a data race. After a race-free call to any of the functions, the corresponding **`atomic_flag`** is considered to be initialized, even it had not been before.

4 **EXAMPLE**

```
        atomic_flag guard = ATOMIC_FLAG_INIT;
        atomic_flag guard = { }; // explicit default initializer
```

7.17.8.1 The `atomic_flag_test_and_set` type-generic macros

Synopsis

1
```
        #include <stdatomic.h>
        _Bool atomic_flag_test_and_set(volatile atomic_flag *object);
        _Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,
            memory_order order);
        bool atomic_flag_test_and_set(A *object);
        bool atomic_flag_test_and_set_explicit(A *object, memory_order order);
```

353) Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the **`atomic_flag`** type is the minimum type needed to conform to this clause that must be indivisible when communicating with a signal handler.

Description

2 Atomically places the atomic flag pointed to by `object` in the set state and returns the value corresponding to the immediately preceding state. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 The **`atomic_flag_test_and_set`** type-generic macros return the value that corresponds to the state of the atomic flag immediately before the effects, if any. The return value ~~true~~ **`true`** corresponds to the set state and the return value ~~false~~ **`false`** corresponds to the clear state. If the flag had not been initialized prior to the call an unspecified value is returned.

7.17.8.2 The `atomic_flag_clear` generic functions

Synopsis

1
```
#include <stdatomic.h>
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *object,
    memory_order order);
void atomic_flag_clear(A *object);
void atomic_flag_clear_explicit(A *object, memory_order order);
```

Description

2 The `order` argument shall not be **`memory_order_acquire`** nor **`memory_order_acq_rel`**. Atomically places the atomic flag pointed to by `object` into the clear state. Memory is affected according to the value of `order`.

Returns

3 The **`atomic_flag_clear`** generic functions return no value.

7.18 Boolean type and values `<stdbool.h>`

1 The obsolescent header `<stdbool.h>` ~~defines four macros.~~

~~expands to **_Bool**.~~

~~Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros bool, **true**, and **false**.~~contains no definitions.

7.19 Common definitions <stddef.h>

1 The obsolescent header `<stddef.h>` ~~defines the following macros and declares the following types. Some are also defined in other headers, as noted in their respective subclauses.~~

~~which expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the subobject (designated by *member-designator*), from the beginning of any object of type *type*. The type and member designator shall be such that given then the expression &(t. *member-designator*) evaluates to an address constant. If the specified *type* defines a new type or if the specified member is a bit-field, the behavior is undefined.~~

~~**Recommended practice**~~

~~The types used for **size_t** and **ptrdiff_t** should not have an integer conversion rank greater than that of **signed long int** unless the implementation supports objects large enough to make this necessary~~provides no content.

7.20 Integer types `<stdint.h>`

1 The header `<stdint.h>` declares sets of integer types having specified widths, and defines corresponding sets of macros.[354] It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

2 Types are defined in the following categories:

- integer types having certain exact widths;
- integer types having at least certain specified widths;
- fastest integer types having at least certain specified widths;
- integer types wide enough to hold pointers to objects;
- integer types having greatest width.

(Some of these types may denote the same type.)

3 Corresponding macros specify limits of the declared types and construct suitable constants.

4 For each type described herein that the implementation provides,[355] `<stdint.h>` shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, `<stdint.h>` shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as "required", but need not provide any of the others (described as "optional").

5 The feature test macro ~~`__STDC_VERSION_STDINT_H__`~~ `__CORE_VERSION_STDINT_H__` expands to the token *202005*`L`.

7.20.1 Integer types

1 When typedef names differing only in the absence or presence of the initial `u` are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

2 In the following descriptions, the symbol N represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

7.20.1.1 Minimum-width integer types

1 The typedef name **`int_least`*N*`_t`** designates a signed integer type with a width of at least N, such that no signed integer type with lesser size has at least the specified width. Thus, **`int_least32_t`** denotes a signed integer type with a width of at least 32 bits.

2 The typedef name **`uint_least`*N*`_t`** designates an unsigned integer type with a width of at least N, such that no unsigned integer type with lesser size has at least the specified width. Thus, **`uint_least16_t`** denotes an unsigned integer type with a width of at least 16 bits.

3 If the macro **`INT_LEAST_WIDTH_MAX`** is not defined, the implementation shall provide these types for all $N = 2^w$ with $3 \leq w$. If **`INT_LEAST_WIDTH_MAX`** is defined it shall be an integer constant expression suitable for use in **`#if`** preprocessing directives and be a power of two $N_0 = 2^{w_0}$ with $6 \leq w_0$, shall be at least **`INT_BITFIELD_MAX`**, and the types shall be provided for all $N = 2^w$ with $3 \leq w \leq w_0$. All other types of this form are optional.

4 Thus, the following types are required:

`int_least8_t`	**`uint_least8_t`**
`int_least16_t`	**`uint_least16_t`**
`int_least32_t`	**`uint_least32_t`**
`int_least64_t`	**`uint_least64_t`**

[354] See "future library directions" (7.32.9).

[355] Some of these types might denote implementation-defined extended integer types.

7.20.1.2 Exact-width integer types

1 The typedef name **int***N***_t** designates a signed integer type with width N and no padding bits. Thus, **int8_t** denotes such a signed integer type with a width of exactly 8 bits.

2 The typedef name **uint***N***_t** designates an unsigned integer type with width N and no padding bits. Thus, uint24_t denotes such an unsigned integer type with a width of exactly 24 bits.

3 ~~These types are optional. However, if~~ If an implementation provides integer types with ~~widths of 8, 16, 32, or 64 bits,~~ width N and no padding bits, it shall define the corresponding typedef names~~.~~; it needs not to provide any such type other than for the special case of N being **CHAR_BIT** for which the types shall be **signed char** and **unsigned char**, respectively.

4 ~~The typedef name **int_least***N***_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width.Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits~~The macros **intwidth**(M) and **uintwidth**(M) designate signed and unsigned integer types, respectively, that provide representations with width M.[356] M shall be an integer constant expression that is strictly positive and, if that is defined, less than or equal to **INT_LEAST_WIDTH_MAX**. For a given valid value M let N be the least value $M \leq N$ such that the types **int_least***N***_t** and **uint_least***N***_t** are provided.

5 ~~The typedef name **uint_least**~~If the types **int***N***_t** and **uint***N***_t** are provided and $M \equiv N$, the types **intwidth**(M) and **uintwidth**(M) designate these types, respectively.

6 ~~designates an unsigned integer~~ Otherwise, let I_N be **int***N***_t** and U_N be **uint***N***_t**, if they are provided, or **int_least***N***_t** and **uint_least***N***_t**, if it is not, and let s be **sizeof**(I_N). Then, **intwidth**(M) and **uintwidth**(M) designate implementation-defined pairs of extended signed and unsigned integer types with a size of s.[357] If an lvalue with such a type undergoes an lvalue conversion or if a value of such a type occurs as the operand of any operator the value is first converted to I_N or U_N, respectively.

7 Analogous rules for signed overflow and unsigned wrap-around apply as in 6.3.1.3;[358] if a value of any integer type is converted to **intwidth**(M) and the value is not representable with a precision of $M-1$, the implementation-defined rules for signed overflow apply; if a value of any integer type is converted to **uintwidth**(M) and the value is not representable with a width of ~~at least~~ M, the value is reduced modulo 2^M.

Recommended practice

8 Implementations should provide all types **int***N***_t**, ~~such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.~~ **uint***N***_t**, **int_least***N***_t**, or **uint_least***N***_t** where N is a multiple of **CHAR_BIT** for which they have suitable standard or extended integer types of size N/**CHAR_BIT**. The minimum-width and exact-width types should only differ if this is unavoidable, that is if for a given $N = 2^w$ with $3 \leq w \leq 6$ there are no exact-width types.

~~The following types are required:~~

9 **NOTE 1** The types designated by **intwidth**(M) and **uintwidth**(M) can be implemented with minimal support by defining **INT_LEAST_WIDTH_MAX** to 64 and by mapping the representations of the types to their **int***N***_t**, **uint***N***_t**, **int_least***N***_t**, or **uint_least***N***_t** counterparts as stipulated by the rules above. Additionally, only the conversion to **uintwidth**(M) has to be implemented as modulo 2^M; conversion to **intwidth**(M) could even be left alone. All other operations then simply follow from the other conversion rules.

~~**int_least8_t**~~
~~**int_least16_t**~~
~~**int_least32_t**~~
~~**int_least64_t**~~
~~**uint_least8_t**~~

[356] But other than the exact width types, even when respecting the rules given in the following, they may have padding bits.

[357] So these types then have $s \times$ **CHAR_BIT** $- M$ padding bits.

[358] Because of the other conversion rules given above, such conversions can only occur explicitly (by a cast), for assignment and for function arguments.

~~uint_least16_t~~
~~uint_least32_t~~
~~uint_least64_t~~ On the other hand, implementations that have support for such types may implement them with much more details, as long as they behave as-if specified as above. In particular, if operations on them can be performed by using special hardware registers, they then should be used as the the corresponding types **int***N***_t**, **uint***N***_t**, **int_least***N***_t**, or **uint_least***N***_t**, where applicable. Such type (or representation) mapping not withstanding, if these are then narrow integer types, the promotion rules still apply.

~~All other~~

11 **NOTE 2** The types ~~of this form are optional.~~ designated by **intwidth**(1) and **uintwidth**(1) are quite particular; **intwidth**(1) only has a sign bit and has values 0 and -1. **uintwidth**(1) only has values 0 and 1 but it is not the same type as **bool**, because conversion from other integer types is not a Boolean conversion, but models the parity of the converted value.

7.20.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[359] to operate with among all integer types that have at least the specified width.

2 The typedef name **int_fast***N***_t** designates the fastest signed integer type with a width of at least *N*. The typedef name **uint_fast***N***_t** designates the fastest unsigned integer type with a width of at least *N*.

3 The following types are required:

```
int_fast8_t      uint_fast8_t
int_fast16_t     uint_fast16_t
int_fast32_t     uint_fast32_t
int_fast64_t     uint_fast64_t
```

All other types of this form are optional.

7.20.1.4 Integer types capable of holding object pointers

1 The following type designates a signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

7.20.1.5 Greatest-width integer types

1 The following type designates a signed integer type capable of representing any value of any signed integer type:

```
intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

```
uintmax_t
```

These types are required.

[359]The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

7.20.2 Widths of specified-width integer types

1 The following object-like macros specify the width of the types declared in `<stdint.h>`. Each macro name corresponds to a similar type name in 7.20.1.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **`#if`** preprocessing directives. Its implementation-defined value shall be equal to or greater than the value given below, except where stated to be exactly the given value. An implementation shall define only the macros corresponding to those typedef names it actually provides;[360)] if **`INT_LEAST_WIDTH_MAX`** is defined, see 7.20.1.1, these macros shall be defined for all N where the corresponding types are defined. If it is not defined, these macros shall be defined for all $64 \leq N$ where the corresponding types are defined.

7.20.2.1 Width of exact-width integer types

1

`INT`*N*`_WIDTH`	exactly N
`UINT`*N*`_WIDTH`	exactly N

7.20.2.2 Width of minimum-width integer types

1

`INT_LEAST`*N*`_WIDTH`	exactly `UINT_LEAST`*N*`_WIDTH`
`UINT_LEAST`*N*`_WIDTH`	N

7.20.2.3 Width of fastest minimum-width integer types

1

`INT_FAST`*N*`_WIDTH`	exactly `UINT_FAST`*N*`_WIDTH`
`UINT_FAST`*N*`_WIDTH`	N

7.20.2.4 Width of integer types capable of holding object pointers

1

`INTPTR_WIDTH`	exactly `UINTPTR_WIDTH`
`UINTPTR_WIDTH`	16

7.20.2.5 Width of greatest-width integer types

1

`INTMAX_WIDTH`	exactly `UINTMAX_WIDTH`
`UINTMAX_WIDTH`	64

7.20.3 Width of other integer types

1 The following object-like macros specify the width of integer types corresponding to types defined in other standard headers.

2 Each instance of these macros shall be replaced by a constant expression suitable for use in **`#if`** preprocessing directives. Its implementation-defined value shall be equal to or greater than the corresponding value given below. An implementation shall define only the macros corresponding to those typedef names it actually provides.[361)]

7.20.3.1 Width of `ptrdiff_t`

1

`PTRDIFF_WIDTH`	17

7.20.3.2 Width of `sig_atomic_t`

1

`SIG_ATOMIC_WIDTH`	8

360) The exact-width and pointer-holding integer types are optional.

361) A freestanding implementation need not provide all of these types.

7.20.3.3 Width of size_t

1
```
SIZE_WIDTH                        16
```

7.20.3.4 Width of wchar_t

1
```
WCHAR_WIDTH                       8
```

7.20.3.5 Width of wint_t

1
```
WINT_WIDTH                        16
```

7.20.4 Macros for integer constants

1 The following function-like macros expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in `<stdint.h>`. Each macro name corresponds to a similar type name in 7.20.1.1 or 7.20.1.5.

2 The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

3 Each invocation of one of these macros shall expand to an integer constant expression suitable for use in **#if** preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

7.20.4.1 Macros for minimum-width integer constants

1 The macro **INT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **int_least***N***_t**. The macro **UINT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **uint_least***N***_t**. For example, if **uint_least64_t** is a name for the type **unsigned long long int**, then **UINT64_C**(0x123) might expand to the integer constant 0x123ULL.

7.20.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type **intmax_t**:

```
INTMAX_C(value)
```

The following macro expands to an integer constant expression having the value specified by its argument and the type **uintmax_t**:

```
UINTMAX_C(value)
```

7.20.5 Maximal and minimal values of integer types

1 For all integer types for which there is a macro with suffix **_WIDTH** holding the width, maximum macros with suffix **_MAX** and, for all signed types, minimum macros with suffix **_MIN** are defined as by 5.2.4.2. If it is unspecified if a type is signed or unsigned and the implementation has it as an unsigned type, a minimum macro with extension **_MIN**, and value 0 of the corresponding type is defined.

7.21 Input/output <stdio.h>

7.21.1 Introduction

1 The header <stdio.h> defines several macros, and declares ~~three~~ two types and many functions for performing input and output.

2 The types declared are

```
FILE
```

which is an opaque object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

```
fpos_t
```

which is a complete object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.[362)]

3 The macros are ~~NULL (described in 7.19);~~

```
_IOFBF
_IOLBF
_IONBF
```

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

```
BUFSIZ
```

which expands to an integer constant expression that is the size of the buffer used by the **setbuf** function;

```
EOF
```

which expands to an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

```
FOPEN_MAX
```

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

```
FILENAME_MAX
```

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened or, if the implementation imposes no practical limit on the length of file name strings, the recommended size of an array intended to hold a file name string;[363)]

```
L_tmpnam
```

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

---

362) Although its representation is unspecified, here, the type **fpos_t** is not opaque and objects of that type can be copied.

363) Of course, file name string contents are subject to other system-specific constraints; therefore *all* possible strings of length **FILENAME_MAX** cannot be expected to be opened successfully.

```
SEEK_CUR
SEEK_END
SEEK_SET
```

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

```
TMP_MAX
```

which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the **tmpnam** function;

```
stderr
stdin
stdout
```

which are expressions of type "pointer to **FILE**" that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

4 The header `<wchar.h>` declares a number of functions useful for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of "generalized" multibyte characters, as described further in 7.21.3.

5 The input/output functions are given the following collective terms:

— The *wide character input functions* — those functions described in 7.29 that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vfwscanf**, and **vwscanf**.

— The *wide character output functions* — those functions described in 7.29 that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, and **vwprintf**.

— The *wide character input/output functions* — the union of the **ungetwc** function, the wide character input functions, and the wide character output functions.

— The *byte input/output functions* — those functions described in this subclause that perform input/output: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vfscanf**, **vprintf**, and **vscanf**.

Forward references: files (7.21.3), the **fseek** function (7.21.9.2), streams (7.21.2), the **tmpnam** function (7.21.4.4), `<wchar.h>` (7.29).

7.21.2 Streams

1 Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.[364)]

2 A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a

[364)] An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of ~~characters~~ bytes that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null ~~characters~~ bytes appended to the end of the stream.

4 Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)[365]

5 Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

6 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

7 Each stream has an associated lock that is used to prevent data races when multiple threads of execution access a stream, and to restrict the interleaving of stream operations performed by multiple threads. Only one thread may hold this lock at a time. The lock is reentrant: a single thread may hold the lock multiple times at a given time.

8 All functions that read, write, position, or query the position of a stream lock the stream before accessing it. They release the lock associated with the stream when the access is complete.

Environmental limits

9 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Forward references: the **freopen** function (7.21.5.4), the **fwide** function (7.29.3.5), **mbstate_t** (7.29.1), the **fgetpos** function (7.21.9.1), the **fsetpos** function (7.21.9.3).

7.21.3 Files

1 A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (~~character~~

[365]The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

byte number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.21.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is *unbuffered*, ~~characters~~ bytes are intended to appear from the source or at the destination as soon as possible. Otherwise ~~characters~~ bytes may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, ~~characters~~ bytes are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a text stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, ~~characters~~ bytes are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **setbuf** and **setvbuf** functions.

4 A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

5 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

6 The address of the **FILE** object used to control a stream may be significant~~; a copy~~. The implementation may have **FILE** as complete or incomplete object type, but as for any opaque object type, **FILE** objects are not modifiable lvalues and can therefore not appear as the first operand of an assignment. Passing a byte-copy of a **FILE** object ~~need not serve in place of the original~~as **FILE**∗ parameter to any of the library functions has undefined behavior.

7 At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

9 Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

— Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

— A file need not begin nor end in the initial shift state.[366)]

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

[366)]Setting the file position indicator to end-of-file, as with `fseek(file, 0, SEEK_END)`, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

12 The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object. The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

13 In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

14 An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized) multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Environmental limits

15 The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Forward references: the **exit** function (7.22.5.4), the **fgetc** function (7.21.7.1), the **fopen** function (7.21.5.3), the **fputc** function (7.21.7.3), the **setbuf** function (7.21.5.5), the **setvbuf** function (7.21.5.6), the **fgetwc** function (7.29.3.1), the **fputwc** function (7.29.3.3), conversion state (7.29.6), the **mbrtowc** function (7.29.6.3.2), the **wcrtomb** function (7.29.6.3.3).

7.21.4 Operations on files

7.21.4.1 The remove function

Synopsis

1
```
#include <stdio.h>
int remove(const char *filename);
int remove(const char *filename) [[ core::modifies(fopen) ]];
```

Description

2 The **remove** function causes the file whose name is the string pointed to by filename to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Returns

3 The **remove** function returns zero if the operation succeeds, nonzero if it fails.

7.21.4.2 The rename function

Synopsis

1
```
#include <stdio.h>
int rename(const char *old, const char *new);
int rename(const char *old, const char *new) [[ core::modifies(fopen) ]];
```

Description

2 The **rename** function causes the file whose name is the string pointed to by old to be henceforth known by the name given by the string pointed to by new. The file named old is no longer accessible by that name. If a file named by the string pointed to by new exists prior to the call to the **rename** function, the behavior is implementation-defined.

Returns

3 The **rename** function returns zero if the operation succeeds, nonzero if it fails,[367] in which case if the file existed previously it is still known by its original name.

7.21.4.3 The tmpfile function

Synopsis

1
```
        #include <stdio.h>
        FILE *tmpfile(void);
        FILE *tmpfile(void) [[core::modifies(fopen)]];
```

Description

2 The **tmpfile** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode.

Recommended practice

3 It should be possible to open at least **TMP_MAX** temporary files during the lifetime of the program (this limit may be shared with **tmpnam**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

4 The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Forward references: the **fopen** function (7.21.5.3).

7.21.4.4 The tmpnam function

Synopsis

1
```
#include <stdio.h>
        char *tmpnam(char *s);
char * [[core::alias]] tmpnam(char *s) [[core::alias(s), core::modifies(fopen)]];
```

Description

2 The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.[368] The function is potentially capable of generating at least **TMP_MAX** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

3 The **tmpnam** function generates a different string each time it is called.

4 Calls to the **tmpnam** function with a null pointer argument may introduce data races with each other. The implementation shall behave as if no library function calls the **tmpnam** function.

Returns

5 If no suitable string can be generated, the **tmpnam** function returns a null pointer. Otherwise, if the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the **tmpnam** function may modify the same object). If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam** **char**s; the **tmpnam** function writes its result in that array and returns the argument as its value.

---

[367] Among the reasons the implementation could cause the **rename** function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

[368] Files created using strings generated by the **tmpnam** function are temporary only in the sense that their names are not expected to collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination.

Environmental limits

6 The value of the macro **TMP_MAX** shall be at least 25.

7.21.5 File access functions

7.21.5.1 The fclose function

Synopsis

1
```
#include <stdio.h>
~~int fclose(FILE *stream);~~
int fclose(FILE *stream) [[core::modifies(fopen)]];
```

Description

2 A successful call to the **fclose** function causes the stream pointed to by stream to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether or not the call succeeds, the stream is disassociated from the file and any buffer set by the **setbuf** or **setvbuf** function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

3 The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

7.21.5.2 The fflush function

Synopsis

1
```
#include <stdio.h>
~~int fflush(FILE *stream);~~
int fflush(FILE *stream) [[core::modifies(fopen)]];
```

Description

2 If stream points to an output stream or an update stream in which the most recent operation was not input, the **fflush** function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

3 If stream is a null pointer, the **fflush** function performs this flushing action on all streams for which the behavior is defined above.

Returns

4 The **fflush** function sets the error indicator for the stream and returns **EOF** if a write error occurs, otherwise it returns zero.

Forward references: the **fopen** function (7.21.5.3).

7.21.5.3 The fopen function

Synopsis

1
```
#include <stdio.h>
~~FILE *fopen(const char * restrict filename, const char * restrict mode);~~

FILE *fopen(const char * [[core::noalias]] filename, const char * [[core::noalias]] mode)
              [[core::modifies(fopen)]];
```

Description

2 The **fopen** function opens the file whose name is the string pointed to by filename, and associates a stream with it.

3 The argument mode points to a string. If the string is one of the following, the file is open in the

indicated mode. Otherwise, the behavior is undefined.[369)]

`r`	open text file for reading
`w`	truncate to zero length or create text file for writing
`wx`	create text file for writing
`a`	append; open or create text file for writing at end-of-file
`rb`	open binary file for reading
`wb`	truncate to zero length or create binary file for writing
`wbx`	create binary file for writing
`ab`	append; open or create binary file for writing at end-of-file
`r+`	open text file for update (reading and writing)
`w+`	truncate to zero length or create text file for update
`w+x`	create text file for update
`a+`	append; open or create text file for update, writing at end-of-file
`r+b` *or* `rb+`	open binary file for update (reading and writing)
`w+b` *or* `wb+`	truncate to zero length or create binary file for update
`w+bx` *or* `wb+x`	create binary file for update
`a+b` *or* `ab+`	append; open or create binary file for update, writing at end-of-file

4 Opening a file with read mode (`'r'` as the first character in the `mode` argument) fails if the file does not exist or cannot be read.

5 Opening a file with exclusive mode (`'x'` as the last character in the `mode` argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.

6 Opening a file with append mode (`'a'` as the first character in the `mode` argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **`fseek`** function. In some implementations, opening a binary file with append mode (`'b'` as the second or third character in the above list of `mode` argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

7 When a file is opened with update mode (`'+'` as the second or third character in the above list of `mode` argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **`fflush`** function or to a file positioning function (**`fseek`**, **`fsetpos`**, or **`rewind`**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

8 When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

9 The **`fopen`** function returns a pointer to the object controlling the stream. If the open operation fails, **`fopen`** returns a null pointer.

Forward references: file positioning functions (7.21.9).

7.21.5.4 The `freopen` function

[369)] If the string begins with one of the above sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in 7.21.2).

Synopsis

1

```
#include <stdio.h>
FILE *freopen(const char * restrict filename, const char * restrict mode,
     FILE * restrict stream);

FILE *freopen(const char *[[core::noalias]] filename, const char *[[core::noalias]] mode,
     FILE *[[core::noalias]] stream) [[core::modifies(fopen)]];
```

Description

2 The **freopen** function opens the file whose name is the string pointed to by `filename` and associates the stream pointed to by `stream` with it. The `mode` argument is used just as in the **fopen** function.[370)]

3 If `filename` is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by `mode`, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

4 The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Returns

5 The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of `stream`.

7.21.5.5 The setbuf function

Synopsis

1

```
#include <stdio.h>
void setbuf(FILE * restrict stream, char * restrict buf);
void setbuf(FILE *[[core::noalias]] stream, char *[[core::noalias]] buf);
```

Description

2 Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for `mode` and **BUFSIZ** for `size`, or (if `buf` is a null pointer), with the value **_IONBF** for `mode`.

Returns

3 The **setbuf** function returns no value.

Forward references: the **setvbuf** function (7.21.5.6).

7.21.5.6 The setvbuf function

Synopsis

1

```
#include <stdio.h>
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);

int setvbuf(FILE *[[core::noalias]] stream, char *[[core::noalias]] buf, int mode, size_t size);
```

Description

2 The **setvbuf** function may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument `mode` determines how `stream` will be buffered, as follows:

_IOFBF causes input/output to be fully buffered;

370)The primary use of the **freopen** function is to change the file associated with a standard text stream (**stderr**, **stdin**, or **stdout**), as those identifiers need not be modifiable lvalues to which the value returned by the **fopen** function could be assigned.

`_IOLBF` causes input/output to be line buffered;

`_IONBF` causes input/output to be unbuffered.

If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by the `setvbuf` function[371] and the argument `size` specifies the size of the array; otherwise, `size` may determine the size of a buffer allocated by the `setvbuf` function. The contents of the array at any time are indeterminate.

Returns

3 The `setvbuf` function returns zero on success, or nonzero if an invalid value is given for `mode` or if the request cannot be honored.

7.21.6 Formatted input/output functions

1 The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[372]

7.21.6.1 The `fprintf` function

Synopsis

1
```
#include <stdio.h>
int fprintf(FILE * [[ core::noalias ]]  stream, const char * [[ core::noalias ]]  format,
    ...)
       [[ core::modifies(errno) ]] ;
```

Description

2 The `fprintf` function writes output to the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a nonnegative decimal integer.[373]

— An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of bytes to be written for `s` conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

[371] The buffer has to have a lifetime at least as great as the open stream, so not closing the stream before a buffer that has automatic storage duration is deallocated upon block exit results in undefined behavior.
[372] The `fprintf` functions perform writes to memory for the `%n` specifier.
[373] Note that `0` is taken as a flag, not as the beginning of a field width.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **`int`** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag characters and their meanings are:

`-` The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

`+` The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)[374)]

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and `+` flags both appear, the *space* flag is ignored.

`#` The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`signed char`** or **`unsigned char`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`signed char`** or **`unsigned char`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`signed char`** argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`short int`** or **`unsigned short int`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`short int`** or **`unsigned short int`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`short int`** argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long int`** or **`unsigned long int`** argument; that a following `n` conversion specifier applies to a pointer to a **`long int`** argument; that a following `c` conversion specifier applies to a **`wint_t`** argument; that a following `s` conversion specifier applies to a pointer to a **`wchar_t`** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

[374)] The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long long int`** or **`unsigned long long int`** argument; or that a following `n` conversion specifier applies to a pointer to a **`long long int`** argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **`intmax_t`** or **`uintmax_t`** argument; or that a following `n` conversion specifier applies to a pointer to an **`intmax_t`** argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`size_t`** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **`size_t`** argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`ptrdiff_t`** or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a **`ptrdiff_t`** argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`long double`** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

`d,i` The **`int`** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o,u,x,X` The **`unsigned int`** argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f,F` A **`double`** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **`double`** argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A **`double`** argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-char-sequence*`)` — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively.[375)]

`e,E` A **`double`** argument representing a floating-point number is converted in the style *[-]d.ddd*`e`*±dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

[375)]When applied to infinite and NaN values, the `-`, `+`, and *space* flag characters have their usual meaning; the `#` and `0` flag characters have no effect.

`g`,`G` A **`double`** argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of X:

if $P > X \geq -4$, the conversion is with style `f` (or `F`) and precision $P - (X + 1)$.

otherwise, the conversion is with style `e` (or `E`) and precision $P - 1$.

Finally, unless the **#** flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a`,`A` A **`double`** argument representing a floating-point number is converted in the style *[-]*`0x`*h.hhhh*`p`$\pm$*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[376)] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **`FLT_RADIX`** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **`FLT_RADIX`** is not a power of 2, then the precision is sufficient to distinguish[377)] values of type **`double`**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` length modifier is present, the **`int`** argument is converted to an **`unsigned char`**,and the resulting character is written.

If an `l` length modifier is present, the **`wint_t`** argument is converted as if by an `ls` conversion specification with no precision and an argument that points to the initial element of a two-element array of **`wchar_t`**, the first element containing the **`wint_t`** argument to the `lc` conversion specification and the second a null wide character.

`s` If no `l` length modifier is present, the argument shall be a pointer to the initial element of an array of character type.[378)] Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of **`wchar_t`** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by

[376)] Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

[377)] The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **`FLT_RADIX`** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

[378)] No special provisions are made for multibyte characters.

the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[379)]

p The argument shall be a pointer to **void**. The value of the pointer shall be valid or null. It is converted to a sequence of printing characters, in an implementation-defined manner. If the value of the pointer is valid its provenance is henceforth exposed.

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A % character is written. No argument is converted. The complete conversion specification shall be %%.

9 If a conversion specification is invalid, the behavior is undefined.[380)] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For a and A conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For a and A conversions, if **FLT_RADIX** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For e, E, f, F, g, and G conversions, if the number of significant decimal digits is at most the maximum value M of the *T*_**DECIMAL_DIG** macros (defined in <float.h>), then the result should be correctly rounded.[381)] If the number of significant decimal digits is more than M but the source value is exactly representable with M digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having M significant digits; the value of the resultant decimal string D should satisfy $L \le D \le U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

14 The **fprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

15 The number of characters that can be produced by any single conversion shall be at least 4095.

16 **EXAMPLE 1** To print a date and time in the form "Sunday, July 3, 10:02" followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month;    // pointers to strings
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
fprintf(stdout, "pi = %.5f\n", 4 × atan(1.0));
```

[379)]Redundant shift sequences can result if multibyte characters have a state-dependent encoding.
[380)]See "future library directions" (7.32.10).
[381)]For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

17 **EXAMPLE 2** In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter.

18 Given the following wide string with length seven,

```
static wchar_t wstr[] = L"□X□Yabc□Z□W";
```

the seven calls

```
fprintf(stdout, "|1234567890123|\n");
fprintf(stdout, "|%13ls|\n", wstr);
fprintf(stdout, "|%-13.9ls|\n", wstr);
fprintf(stdout, "|%13.10ls|\n", wstr);
fprintf(stdout, "|%13.11ls|\n", wstr);
fprintf(stdout, "|%13.15ls|\n", &wstr[2]);
fprintf(stdout, "|%13lc|\n", (wint_t) wstr[5]);
```

will print the following seven lines:

```
|1234567890123|
|   □X□Yabc□Z□W|
|□X□Yabc□Z     |
|     □X□Yabc□Z|
|   □X□Yabc□Z□W|
|       abc□Z□W|
|            □Z|
```

Forward references: conversion state (7.29.6), the `wcrtomb` function (7.29.6.3.3).

7.21.6.2 The `fscanf` function

Synopsis

1
```
#include <stdio.h>
int fscanf(FILE * [[ core::noalias ]] stream, const char * [[ core::noalias ]] format, ...
    )
      [[ core::modifies(errno) ]] ;
```

Description

2 The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state.The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— An optional assignment-suppressing character `*`.

— An optional decimal integer greater than zero that specifies the maximum field width (in characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* character that specifies the type of conversion to be applied.

4 The **fscanf** function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. The directive never fails.

6 A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive,the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space characters are skipped, unless the specification includes a `[`, `c`, or `n` specifier.[382)]

9 An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[383)] The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **double**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **wchar_t**.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

382) These white-space characters are not counted against a specified field width.

383) **fscanf** pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to **strtod**, **strtol**, etc., are unacceptable to **fscanf**.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`long double`**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **`strtol`** function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to signed integer.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **`strtol`** function with the value 0 for the `base` argument. The corresponding argument shall be a pointer to signed integer.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **`strtoul`** function with the value 8 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **`strtoul`** function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **`strtoul`** function with the value 16 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`a,e,f,g` Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **`strtod`** function. The corresponding argument shall be a pointer to floating.

`c` Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).[384)]

If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the resulting sequence of wide characters.No null wide character is added.

`s` Matches a sequence of non-white-space characters.[384)]

If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[384)]No special provisions are made for multibyte characters in the matching rules used by the `c`, `s`, and `[` conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[384)]

If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches the same implementation-defined set of sequences of characters that may be produced by the `%p` conversion of the **`fprintf`** function. The corresponding argument `ptr` shall be a pointer to a pointer to **`void`**.

- If the input sequence could have been printed from a null pointer value, `*ptr` is assigned a null pointer value.
- Otherwise, if the input sequence could have been printed from a valid pointer x and if the address x currently refers to an exposed storage instance, a valid pointer with address x and the provenance of that storage instance is synthesized in `*ptr`.[385)]
- Otherwise `*ptr` becomes indeterminate.

`n` No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **`fscanf`** function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the **`fscanf`** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

`%` Matches a single `%` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[386)]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

15 Trailing white-space characters(including new-line characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

385)Thus, the constructed pointer value has a valid provenance. Nevertheless, because the original storage instance might be dead and a new storage instance might live at the same address, this provenance can be different from the provenance that gave rise to the print operation.If x can be an address with more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

386)See "future library directions" (7.32.10).

Returns

16 The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 **EXAMPLE 1** The call:

```
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `thompson\0`.

18 **EXAMPLE 2** The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip `0123`, and will assign to `name` the sequence `56\0`. The next character read from the input stream will be `a`.

19 **EXAMPLE 3** To accept repeatedly from **stdin** a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21], item[21];
do {
      count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
      fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
} while (¬feof(stdin) ∧ ¬ferror(stdin));
```

20 If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

21 **EXAMPLE 4** In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to `d1` and the value 3 to `n1`. Because `%n` can never get an input failure, the value of 3 is also assigned to `n2`. The value of `d2` is not affected. The value 1 is assigned to `i`.

22 **EXAMPLE 5** The call:

```
#include <stdio.h>
/* ... */
int n, i;
n = sscanf("foo  %bar  42", "foo%%bar%d", &i);
```

will assign to `n` the value 1 and to `i` the value 42 because input white-space characters are skipped for both the `%` and `d` conversion specifiers.

23 **EXAMPLE 6** In these examples, multibyte characters do have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter, but are only recognized as such when in the alternate shift state. The shift sequences are denoted by ↑ and ↓, in which the first causes entry into the alternate shift state.

24 After the call:

```
#include <stdio.h>
/* ... */
char str[50];
fscanf(stdin, "a%s", str);
```

with the input line:

```
a↑□X□Y↓ bc
```

`str` will contain `↑□X□Y↓\\0` assuming that none of the bytes of the shift sequences (or of the multibyte characters, in the more general case) appears to be a single-byte white-space character.

25 In contrast, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a%ls", wstr);
```

with the same input line, `wstr` will contain the two wide characters that correspond to □X and □Y and a terminating null wide character.

26 However, the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□X↓%ls", wstr);
```

with the same input line will return zero due to a matching failure against the ↓ sequence in the format string.

27 Assuming that the first byte of the multibyte character □X is the same as the first byte of the multibyte character □Y, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□Y↓%ls", wstr);
```

with the same input line, zero will again be returned, but **stdin** will be left with a partially consumed multibyte character.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3), the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.4), conversion state (7.29.6), the **wcrtomb** function (7.29.6.3.3).

7.21.6.3 The printf function

Synopsis

1
```
#include <stdio.h>
int printf(const char * restrict format, ...);
int printf(const char * [[ core::noalias ]] format, ...) [[ core::evaluates(stdout) ]];
```

Description

2 The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Returns

3 The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.4 The scanf function

Synopsis

1
```
#include <stdio.h>
int scanf(const char * restrict format, ...);
int scanf(const char * [[ core::noalias ]] format, ...) [[ core::evaluates(stdin) ]];
```

Description

2 The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Returns

3 The **scanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.5 The snprintf function

Synopsis

1
```
#include <stdio.h>
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);

int snprintf(char * [[ core::noalias ]] s, size_t n, const char * [[ core::noalias ]] format, ...);
```

Description

2 The **snprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by argument s) rather than to a stream. If n is zero, nothing is written, and s may be a null pointer. Otherwise, output characters beyond the n-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **snprintf** function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

7.21.6.6 The `sprintf` function

Synopsis

1
```
#include <stdio.h>
int sprintf(char * restrict s, const char * restrict format, ...);
int sprintf(char * [[core::noalias, core::writethrough]] s, const char * [[core::noalias]] format, ...);
```

Description

2 The **`sprintf`** function is equivalent to **`fprintf`**, except that the output is written into an array (specified by the argument `s`) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **`sprintf`** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.7 The `sscanf` function

Synopsis

1
```
#include <stdio.h>
int sscanf(const char * restrict s, const char * restrict format, ...);
int sscanf(const char * [[core::noalias]] s, const char * [[core::noalias]] format, ...);
```

Description

2 The **`sscanf`** function is equivalent to **`fscanf`**, except that input is obtained from a string (specified by the argument `s`) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **`fscanf`** function. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **`sscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`sscanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.8 The `vfprintf` function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);
int vfprintf(FILE * [[core::noalias]] stream, const char * [[core::noalias]] format, va_list arg)
        [[core::modifies(errno)]];
```

Description

2 The **`vfprintf`** function is equivalent to **`fprintf`**, with the variable argument list replaced by `arg`, which shall have been initialized by the **`va_start`** macro (and possibly subsequent **`va_arg`** calls). The **`vfprintf`** function does not invoke the **`va_end`** macro.[387)]

[387)] As the functions **`vfprintf`**, **`vfscanf`**, **`vprintf`**, **`vscanf`**, **`vsnprintf`**, **`vsprintf`**, and **`vsscanf`** invoke the **`va_arg`** macro, the value of `arg` after the return is indeterminate.

Returns

3 The **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

4 **EXAMPLE** The following shows the use of the **vfprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>

~~void error(char *function_name, char *format, ...)~~
void error(char *function_name, char *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fprintf(stderr, "ERROR in %s: ", function_name);
      // print out remainder of message
      vfprintf(stderr, format, args);
      va_end(args);
}
```

7.21.6.9 The vfscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
~~int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);~~

int vfscanf(FILE *[[core::noalias]] stream, const char *[[core::noalias]] format, va_list arg);
```

Description

2 The **vfscanf** function is equivalent to **fscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf** function does not invoke the **va_end** macro.[387)]

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.10 The vprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
~~int vprintf(const char * restrict format, va_list arg);~~
int vprintf(const char *[[core::noalias]] format, va_list arg)
      [[core::modifies(errno)]];
```

Description

2 The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[387)]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.11 The vscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format, va_list arg);
int vscanf(const char *[[core::noalias]] format, va_list arg);
```

Description

2 The **vscanf** function is equivalent to **scanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf** function does not invoke the **va_end** macro.[387)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.12 The vsnprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>

int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);
int vsnprintf(char *[[core::noalias, core::writethrough]] s, size_t n,
        const char *[[core::noalias]] format, va_list arg);
```

Description

2 The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro.[387)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsnprintf** function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

7.21.6.13 The vsprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s, const char * restrict format, va_list arg);
int vsprintf(char *[[core::noalias, core::writethrough]] s,
        const char *[[core::noalias]] format, va_list arg)
              [[core::modifies(errno)]];
```

Description

2 The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro.[387)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.14 The vsscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
int vsscanf(const char *[[core::noalias]] s, const char *[[core::noalias]] format, va_list arg);
```

Description

2 The **vsscanf** function is equivalent to **sscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsscanf** function does not invoke the **va_end** macro.[387)]

Returns

3 The **vsscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vsscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.7 Character input/output functions

7.21.7.1 The fgetc function

Synopsis

1
```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by stream is not set and a next character is present, the **fgetc** function obtains that character as an **unsigned char** converted to an **int** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the **fgetc** function returns **EOF**. Otherwise, the **fgetc** function returns the next character from the input stream pointed to by stream. If a read error occurs, the error indicator for the stream is set and the **fgetc** function returns **EOF**.[388)]

7.21.7.2 The fgets function

Synopsis

1
```
#include <stdio.h>
char *fgets(char * restrict s, int n, FILE * restrict stream);
char *fgets(char *[[core::noalias, core::writethrough]] s, int n, FILE *[[core::noalias]] stream);
```

Description

2 The **fgets** function reads at most one less than the number of characters specified by n from the stream pointed to by stream into the array pointed to by s. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

[388)] An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

Returns

3 The **fgets** function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.21.7.3 The `fputc` function

Synopsis

1
```
        #include <stdio.h>
        int fputc(int c, FILE *stream);
```

Description

2 The **fputc** function writes the ~~character~~ byte value specified by `c` (converted to an **unsigned char**) to the output stream pointed to by `stream`, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the ~~character~~ byte is appended to the output stream.

Returns

3 The **fputc** function returns the ~~character~~ byte value written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

7.21.7.4 The `fputs` type-generic macro

Synopsis

1
```
#include <stdio.h>
        int fputs(const char * restrict s, FILE * restrict stream);
int fputs(A x, FILE *[[core::noalias]] stream)
          [[core::stateless]];
int fputs(A x, FILE *[[core::noalias]] stream, A x, UT baseflags)
          [[core::stateless]];
int fputs(A x, FILE *[[core::noalias]] stream, A x, UT baseflags,
                const char*[[core::noalias]] glue)
          [[core::stateless]];
```

Contraints

2 The corresponding arguments shall have same types and fulfill constraints as for the **totext** type-generic macro. If *A* is a pointer type and according to the rules of **totext** the pointed-to object will be accessed, a **core::noalias** attribute for `x` shall be implied. If the corresponding conversion implies a wide-character or wide-string to multi-byte conversion, **core::modifies(errno)** and **core::evaluates(locale)** attributes shall be implied.

Description

3 The **fputs** type-generic macro writes the ~~string pointed to by s~~ textual representation of `x` as of **totext** to the stream pointed to by `stream`. The terminating null character is not written.

4 The whole write operation of the **fputs** type-generic macro appears indivisible and the output of any such operation appears contiguously on the stream, even in the presence of multiple threads.

5 The interactions with **locale** and **errno** are the same as for **totext**, as are the rules for the exposure of a storage instance for which the textual representation of a pointer value is written to `stream`.

6 An evaluation of the **fputs** type-generic macro without arguments results in a function pointer of the type

```
int (*)(const char*[[core::noalias]] x, FILE *[[core::noalias]] stream)
```

When it is called with appropriate arguments, such a function pointer behaves like the macro.

Returns

7 ~~The~~ If called with arguments, the **fputs** type-generic macro returns **EOF** if ~~a write~~ an error occurs; otherwise it returns a nonnegative value. An error can be a write error, an allocation error or an encoding error. Encoding errors occur as if for a call **totext**(0, **nullptr**, x, baseflags, glue) (or similar without the optional arguments) and in that case **errno** is set to **EILSEQ**.

8 If evaluated without arguments, the **fputs** type-generic macro returns a function pointer as indicated above.

Forward references: text conversion functions (7.22.2).

9 **NOTE 1** A specializations if A is a pointer to string could be as different as the following two lambdas, only that in addition the associated lock of stream is taken by the current thread and released afterwards.

```
[](char* x, FILE *[[core::noalias]] stream) {
    int ret = 1;
    for (; *x ∧ ret ≠ EOF; ++x) {
        ret = fputc(*x, stream);
    }
    return ret;
}
```

```
[](char* x, FILE *[[core::noalias]] stream) {
    int ret = EOF;
    size_t n = strlen(x) + 1;
    int ret = (n ∧ (fwrite(buffer, n-1, 1, stream) ≡ n-1)) 1 ? EOF;
    return ret;
}
```

Each of the two specializations would avoid unnecessary copy operations for that case.

7.21.7.5 The getc function

Synopsis

1
```
#include <stdio.h>
int getc(FILE *stream);
```

Description

2 The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expression with side effects.

Returns

3 The **getc** function returns the next ~~character~~ byte from the input stream pointed to by stream. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

7.21.7.6 The getchar function

Synopsis

1
```
#include <stdio.h>
int getchar(void);
int getchar(void) [[core::evaluates(stdin)]];
```

Description

2 The **getchar** function is equivalent to **getc** with the argument **stdin**.

Returns

3 The **getchar** function returns the next ~~character~~ byte from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

7.21.7.7 The `putc` function

Synopsis

1
```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description

2 The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate `stream` more than once, so that argument should never be an expression with side effects.

Returns

3 The **putc** function returns the ~~character~~ byte value written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

7.21.7.8 The `putchar` function

Synopsis

1
```
#include <stdio.h>
int putchar(int c);
int putchar(int c) [[core::evaluates(stdout)]];
```

Description

2 The **putchar** function is equivalent to **putc** with the second argument **stdout**.

Returns

3 The **putchar** function returns the ~~character~~ byte value written. If a write error occurs, the error indicator for the stream is set and **putchar** returns **EOF**.

7.21.7.9 The `puts` type-generic macro

Synopsis

1
```
#include <stdio.h>
        int puts(const char *s);
int puts(A x)
        [[core::modifies(stdout), core::stateless]];
int puts(A x, UT baseflags)
        [[core::modifies(stdout), core::stateless]];
int puts(A x, UT baseflags, const char* [[core::noalias]] glue)
        [[core::modifies(stdout), core::stateless]];
```

Contraints

2 The arguments shall have same types and fulfill constraints as for the correponding arguments to the **totext** type-generic macro. If *A* is a pointer type and according to the rules of **totext** the pointed-to object will be accessed, a **core::noalias** attribute for `x` shall be implied. If the corresponding conversion implies a wide-character or wide-string to multi-byte conversion, **core::modifies(errno)** and **core::evaluates(locale)** attributes shall be implied.

Description

3 The **puts** type-generic macro writes the ~~string pointed to by `s`~~ textual representation of `x` as of **totext** without the terminating null character to the stream pointed to by **stdout**, and appends a new-line character to the output. ~~The terminating null character is not written.~~

4 The whole write operation of the **puts** type-generic macro appears indivisible and the output of any such operation appears contiguously on the standard output stream, even in the presence of multiple threads.

5 The interactions with **locale** and **errno** are the same as for **totext**, as are the rules for the exposure of a storage instance for which the textual representation of a pointer value is written to the standard output stream.

6 An evaluation of the **puts** type-generic macro without arguments results in a function pointer of the type

```
int (*)(const char* x);
```

When it is called with an appropriate argument, such a function pointer behaves like the macro.

Returns

7 ~~The~~ If called with arguments, the **puts** type-generic macro returns **EOF** if ~~a write~~ an error occurs; otherwise it returns a nonnegative value. An error can be a write error, an allocation error or an encoding error. Encoding errors occur as if for a call **totext**(0, **nullptr**, x, baseflags, glue) (or similar without the optional arguments) and in that case **errno** is set to **EILSEQ**.

8 If evaluated without arguments, the **puts** type-generic macro returns a function pointer as indicated above.

Forward references: text conversion functions (7.22.2).

7.21.7.10 The ungetc function

Synopsis

1
```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

2 The **ungetc** function pushes the ~~character~~ byte value specified by c (converted to an **unsigned char**) back onto the input stream pointed to by stream. Pushed-back ~~characters~~ bytes will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

3 One ~~character~~ byte of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of c equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

5 A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the ~~characters~~ bytes were pushed back.[389] For a text stream, the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back ~~characters~~ bytes (interpreted as characters) are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it is indeterminate after the call.[390]

Returns

6 The **ungetc** function returns the ~~character~~ byte value pushed back after conversion, or **EOF** if the operation fails.

Forward references: file positioning functions (7.21.9).

7.21.8 Direct input/output functions

7.21.8.1 The fread function

Synopsis

1
```
#include <stdio.h>
```

[389] Note that a file positioning function could further modify the file position indicator after discarding any pushed-back ~~characters~~bytes.

[390] See "future library directions" (7.32.10).

```
        size_t fread(void * restrict ptr, size_t size, size_t nmemb,
                FILE * restrict stream);

     size_t fread(void * [[core::noalias, core::writethrough]] ptr, size_t size, size_t nmemb,
                FILE * [[core::noalias]] stream);
```

Description

2 The **fread** function reads, into the array pointed to by `ptr`, up to `nmemb` ~~elements~~ objects whose size is specified by `size`, from the stream pointed to by `stream`. For each object, `size` calls are made to the **fgetc** function and the ~~results~~ resulting byte values are stored, in the order read, in ~~an array of **unsigned char** exactly overlaying~~ the representation array of the object. The file position indicator for the stream (if defined) is advanced by the number of ~~characters~~ bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns

3 The **fread** function returns the number of elements successfully read, which may be less than `nmemb` if a read error or end-of-file is encountered. If `size` or `nmemb` is zero, **fread** returns zero and the contents of the array and the state of the stream remain unchanged.

7.21.8.2 The fwrite function

Synopsis

1
```
        #include <stdio.h>
        size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
                FILE * restrict stream);
        size_t fwrite(const void * [[core::noalias]] ptr, size_t size, size_t nmemb,
                FILE * [[core::noalias]] stream);
```

Description

2 The **fwrite** function writes, from the array pointed to by `ptr`, up to `nmemb` ~~elements~~ objects whose size is specified by `size`, to the stream pointed to by `stream`. For each object, `size` calls are made to the **fputc** function, taking the representation byte values (in order) from ~~an array of **unsigned char** exactly overlaying~~ the object. The file position indicator for the stream (if defined) is advanced by the number of ~~characters~~ bytes successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

3 If the object (or part thereof) corresponding to the first `size*nmemb` bytes referred by `ptr` contains a valid pointer value with provenance `x`, the **fwrite** function exposes `x`.

Returns

4 The **fwrite** function returns the number of elements successfully written, which will be less than `nmemb` only if a write error is encountered. If `size` or `nmemb` is zero, **fwrite** returns zero and the state of the stream remains unchanged.

7.21.9 File positioning functions

7.21.9.1 The fgetpos function

Synopsis

1
```
        #include <stdio.h>
        int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
        int fgetpos(FILE * [[core::noalias]] stream, fpos_t * [[core::noalias]] pos)
                [[core::modifies(errno)]];
```

Description

2 The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by `stream` in the object pointed to by `pos`. The values stored contain

unspecified information usable by the `fsetpos` function for repositioning the stream to its position at the time of the call to the `fgetpos` function.

Returns

3 If successful, the `fgetpos` function returns zero; on failure, the `fgetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

Forward references: the `fsetpos` function (7.21.9.3).

7.21.9.2 The `fseek` function

Synopsis

1
```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

2 The **`fseek`** function sets the file position indicator for the stream pointed to by `stream`. If a read or write error occurs, the error indicator for the stream is set and **`fseek`** fails.

3 For a binary stream, the new position, measured in ~~characters~~ bytes from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`. The specified position is the beginning of the file if `whence` is **`SEEK_SET`**, the current value of the file position indicator if **`SEEK_CUR`**, or end-of-file if **`SEEK_END`**. A binary stream need not meaningfully support **`fseek`** calls with a `whence` value of **`SEEK_END`**.

4 For a text stream, either `offset` shall be zero, or `offset` shall be a value returned by an earlier successful call to the **`ftell`** function on a stream associated with the same file and `whence` shall be **`SEEK_SET`**.

5 After determining the new position, a successful call to the **`fseek`** function undoes any effects of the **`ungetc`** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful **`fseek`** call, the next operation on an update stream may be either input or output.

Returns

6 The **`fseek`** function returns nonzero only for a request that cannot be satisfied.

Forward references: the **`ftell`** function (7.21.9.4).

7.21.9.3 The `fsetpos` function

Synopsis

1
```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos)
        [[core::modifies(errno)]];
```

Description

2 The **`fsetpos`** function sets the **`mbstate_t`** object (if any) and file position indicator for the stream pointed to by `stream` according to the value of the object pointed to by `pos`, which shall be a value obtained from an earlier successful call to the **`fgetpos`** function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and **`fsetpos`** fails.

3 A successful call to the **`fsetpos`** function undoes any effects of the **`ungetc`** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful **`fsetpos`** call, the next operation on an update stream may be either input or output.

Returns

4 If successful, the **`fsetpos`** function returns zero; on failure, the **`fsetpos`** function returns nonzero and stores an implementation-defined positive value in **`errno`**.

7.21.9.4 The `ftell` function

Synopsis

1
```
#include <stdio.h>
long int ftell(FILE *stream);
long int ftell(FILE *stream) [[core::modifies(errno)]];
```

Description

2 The **`ftell`** function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of ~~characters~~ bytes from the beginning of the file. For a text stream, its file position indicator contains unspecified information, usable by the **`fseek`** function for returning the file position indicator for the stream to its position at the time of the **`ftell`** call; the difference between two such return values is not necessarily a meaningful measure of the number of ~~characters~~ bytes (interpreted as characters) written or read.

Returns

3 If successful, the **`ftell`** function returns the current value of the file position indicator for the stream. On failure, the **`ftell`** function returns -1L and stores an implementation-defined positive value in **`errno`**.

7.21.9.5 The `rewind` function

Synopsis

1
```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

2 The **`rewind`** function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns

3 The **`rewind`** function returns no value.

7.21.10 Error-handling functions

7.21.10.1 The `clearerr` function

Synopsis

1
```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

2 The **`clearerr`** function clears the end-of-file and error indicators for the stream pointed to by `stream`.

Returns

3 The **`clearerr`** function returns no value.

7.21.10.2 The `feof` function

Synopsis

1
```
#include <stdio.h>
int feof(FILE *stream);
```

Description

2 The **`feof`** function tests the end-of-file indicator for the stream pointed to by `stream`.

Returns

3 The **`feof`** function returns nonzero if and only if the end-of-file indicator is set for `stream`.

7.21.10.3 The `ferror` function

Synopsis

1
```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

2 The `ferror` function tests the error indicator for the stream pointed to by `stream`.

Returns

3 The `ferror` function returns nonzero if and only if the error indicator is set for `stream`.

7.21.10.4 The `perror` function

Synopsis

1
```
#include <stdio.h>
void perror(const char *s);
void perror(const char *s) [[core::evaluates(errno, stderr)]];
```

Description

2 The `perror` function maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first (if `s` is not a null pointer and the character pointed to by `s` is not the null character), the string pointed to by `s` followed by a colon (`:`) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the `strerror` function with argument `errno`.

Returns

3 The `perror` function returns no value.

Forward references: the `strerror` function (7.24.6.3).

7.22 General utilities <stdlib.h>

1 The header <stdlib.h> declares ~~five types and several~~ several types and functions of general utility, and defines several macros.[391)]

2 The feature test macro ~~__STDC_VERSION_STDLIB_H__~~ __CORE_VERSION_STDLIB_H__ expands to the token *202005*L.

3 ~~The~~ The obsolescent types declared are **size_t** ~~and~~ **wchar_t** ~~(both described in 7.19), which is a structure type that is the type of the value returned by the **div** function, which is a structure type that is the type of the value returned by the **ldiv** function, and which is a structure type that is the type of the value returned by the **lldiv** function~~**div_t**, **ldiv_t**, and **lldiv_t**, which are structure return types of the obsolescent functions **div**, **ldiv**, and **lldiv**, respectively.

4 The macros defined are

```
EXIT_FAILURE
```

and

```
EXIT_SUCCESS
```

which expand to integer constant expressions that can be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

```
RAND_MAX
```

which expands to an integer constant expression that is the maximum value returned by the **rand** function; and

```
MB_CUR_MAX
```

which expands to a positive integer expression with type **size_t** that is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), which is never greater than **MB_LEN_MAX**.

7.22.1 Numeric conversion functions

1 The functions **atof**, **atoi**, **atol**, and **atoll** need not affect the value of the integer expression **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

2 Functions and macros in this clause do not modify any bytes to which their arguments point. If a pointed-to type of any of their arguments is **volatile** qualified the corresponding rules for **volatile** access apply. The synopsis give descriptions of type-generic macros in terms of supported prototypes. Parameter types are indicated as pointers to type C, which corresponds to a qualified or unqualified character type. The prototype that is chosen for such a specific call to such a macro is determined by the first argument.

3 **NOTE** For each type-generic macro, the C and C++ standards have a function of the same name. Unfortunately, many of they return a pointer to their first argument to the second pointer-to-pointer argument and thereby drop a **const**-qualifier from the pointed-to type of a parameter. Also, the functions can only be called for **volatile** qualified objects, if the qualification is cast away, and thus the load operations that are performed will generally not be conforming to the requirements for **volatile** objects. Applications should prefer to use the type-generic macros to avoid write-privilege escalation on **const** qualified byte arrays.

7.22.1.1 The atof function

Synopsis

1
```
#include <stdlib.h>
double atof(const char *nptr);
double atof(const char *nptr) [[ core::modifies(errno), core::evaluates(locale) ]] ;
```

[391)]See "future library directions" (7.32.11).

Description

2 The **atof** function converts the initial portion of the string pointed to by `nptr` to **double** representation. Except for the behavior on error, it is equivalent to

```
~~strtod(nptr, (char **)NULL)~~
strtod(nptr, nullptr)
```

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.3).

7.22.1.2 The atoi, atol, and atoll functions

Synopsis

1
```
#include <stdlib.h>
~~int atoi(const char *nptr);~~
~~long int atol(const char *nptr);~~
~~long long int atoll(const char *nptr);~~
int atoi(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
long int atol(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
long long int atoll(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by `nptr` to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
~~atoi:  (int)strtol(nptr, (char **)NULL, 10)~~
~~atol:  strtol(nptr, (char **)NULL, 10)~~
~~atoll: strtoll(nptr, (char **)NULL, 10)~~
atoi:  (int)strtol(nptr, nullptr, 10)
atol:  strtol(nptr, nullptr, 10)
atoll: strtoll(nptr, nullptr, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** ~~functions (??~~type-generic macros (7.22.1.4).

7.22.1.3 The strtod, strtof, and strtold type-generic macros

Synopsis

1
```
#include <stdlib.h>
~~double strtod(const char * restrict nptr,~~
~~        char ** restrict endptr);~~
~~float strtof(const char * restrict nptr,~~
~~        char ** restrict endptr);~~
~~long double strtold(const char * restrict nptr,~~
~~        char ** restrict endptr);~~
double strtod(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
float strtof(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
long double strtold(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
```

Constraints

2 The second argument to a call to these macros shall be a null pointer constant or a pointer to the same type as the first argument.

Description

3 The **strtod**, **strtof**, and **strtold** ~~functions~~ type-generic macros convert the initial portion of the string pointed to by `nptr` to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

4 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2;

— a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part as defined in 6.4.4.2;

— `INF` or `INFINITY`, ignoring case

— `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)`, ignoring case in the `NAN` part, where:

n-char-sequence:
digit
nondigit
n-char-sequence digit
n-char-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

5 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated.[392] A character sequence `INF` or `INFINITY` is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)` is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequence is implementation-defined.[393] A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

6 If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, the value resulting from the conversion is correctly rounded.

7 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

[392]It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods could yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

[393]An implementation can use the n-char sequence to determine extra information to be represented in the NaN's significand.

8 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Recommended practice

9 If the subject sequence has the hexadecimal form, **`FLT_RADIX`** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

10 If the subject sequence has the decimal form and at most M significant digits, where M is the maximum value of the *T*`_DECIMAL_DIG` macros (defined in `<float.h>`), the result should be correctly rounded. If the subject sequence D has the decimal form and more than M significant digits, consider the two bounding, adjacent decimal strings L and U, both having M significant digits, such that the values of L, D, and U satisfy $L \le D \le U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.[394)]

Returns

11 The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus **`HUGE_VAL`**, **`HUGE_VALF`**, or **`HUGE_VALL`** is returned (according to the return type and sign of the value), and the value of the macro **`ERANGE`** is stored in **`errno`**. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined.

7.22.1.4 The `strtol`, `strtoll`, `strtoul`, and `strtoull` type-generic macros

Synopsis

1
```
#include <stdlib.h>
long int strtol(
        const char * restrict nptr,
        char ** restrict endptr,
        int base);
long long int strtoll(
        const char * restrict nptr,
        char ** restrict endptr,
        int base);
unsigned long int strtoul(
        const char * restrict nptr,
        char ** restrict endptr,
        int base);
unsigned long long int strtoull(
        const char * restrict nptr,
        char ** restrict endptr,
        int base);
long int strtol(C * [[core::noalias]] nptr,
        C ** [[core::noalias]] endptr,
        int base) [[core::modifies(errno), core::evaluates(locale)]];
long long int strtoll(C * [[core::noalias]] nptr,
        C ** [[core::noalias]] endptr,
        int base)
        [[core::modifies(errno), core::evaluates(locale)]];
unsigned long int strtoul(C * [[core::noalias]] nptr,
        C ** [[core::noalias]] endptr,
        int base)
        [[core::modifies(errno), core::evaluates(locale)]];
unsigned long long int strtoull(C * [[core::noalias]] nptr,
```

394) M is sufficiently large that L and U will usually correctly round to the same internal floating value, but if not will correctly round to adjacent values.

```
        C ** [[ core::noalias ]] endptr,
        int base)
        [[ core::modifies(errno), core::evaluates(locale) ]];
```

Constraints

2 The second argument to a call to these macros shall be a null pointer constant or a pointer to the same type as the first argument.

Description

3 The **strtol**, **strtoll**, **strtoul**, and **strtoull** ~~functions~~ type-generic macros convert the initial portion of the string pointed to by nptr to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling an integer represented in some radix determined by the value of base, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

4 If the value of base is zero, the expected form of the subject sequence is that of an integer constant as described in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

5 The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

6 If the subject sequence has the expected form and the value of base is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

7 In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

8 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Returns

9 The **strtol**, **strtoll**, **strtoul**, and **strtoull** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.22.2 Text conversion functions

1 In the following UT is an unspecified unsigned integer type that is wide enough to hold the *base*, *flags* and *precision* for the **totext** tool:

— The base may have the values 0, 2, 8 or 16.

— Flags may be one of the following identifiers that are integer constant expressions:

identifier	description

`totext_alt`	chose an alternative representation
`totext_char`	force interpretation as characters or strings
`totext_exp`	exponential representation for floating point
`totext_fixed`	fixed point representation for floating
`totext_sign`	force appearance of a sign
`totext_up`	force uppercase letters in number representations

— The precision can be given by calling the macro `totext_precision(`*prec*`)`, where $0 \leq prec < 254$.

2 All these values are bitwise or'ed to form a `baseflags` argument for **`totext`**, below.

7.22.2.1 The `totext` type-generic macro

Synopsis

1
```
#include <stdlib.h>
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x)
     [[core::stateless, core::idempotent]];
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x, UT baseflags)
     [[core::stateless, core::idempotent]];
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x, UT baseflags,
          const char* [[core::noalias]] glue)
     [[core::stateless, core::idempotent]];
```

Constraints

2 The **`totext`** type-generic macro shall not be evaluated without arguments.[395)]

3 Depending on the number of arguments, a call to the **`totext`** type-generic macro behaves as if a function with the indicated prototype is called. For all cases, the type $A^{[x]}$ of an argument to the `x` parameter shall be an arithmetic type, a character type, a wide-character type, an object pointer type or **`nullptr_t`**, or an array thereof, and such that the generic type of $A^{[x]}$ is the scalar type A. In the following, if A is an object pointer type A^x denotes the pointed-to type, otherwise it is the same as A.

4 If an argument for `baseflags` is provided, it shall be an integer constant expression that is the bitwise-or of the values 0, 2, 10, 16, the flags as given above, and a precision as formed above with a value that is not larger than 254. If it is not provided, a value of 0 is assumed. The following constraints apply:

— If the base is different from 0 or the flag **`totext_sign`** is used, A^x shall be an arithmetic type, a character type or a wide-character type.

— If A^x is a floating point type the base shall be 0, 10 or 16.

— If **`totext_exp`** or **`totext_fixed`** are used, A^x shall be a floating point type and the base shall be 0 or 10.

— If **`totext_char`** is used, the base shall be 0. Then, A^x shall be **`bool`**, be a character type or wide-character type, or be an integer type such that there exist a character type or wide-character type with the same width.

5 If an argument to `glue` is provided, it shall be a string literal or **`nullptr`**.

6 If A is not a pointer type, a **`core::unsequenced`** attribute and a **`constexpr`** specifier shall be implied; if it is a pointer type and according to the rules below the pointed-to object will be accessed, a **`core::noalias`** attribute for `x` shall be implied. If the rules for the kind of conversion

[395)]The functionality of the **`totext`** type-generic macro is such that it cannot be converted to function pointers with the same functionality.

as defined below do not imply the storage of the textual representation of a pointer value, the `core::state_transparent` attribute shall be implied;[396] if they imply a wide-character or wide-string to multi-byte conversion, `core::modifies(errno)` and `core::evaluates(locale)` attributes shall be implied.[397]

Description

7 If $n > 1$, the **totext** type-generic macro converts the argument `x` into a null-terminated textual representation no longer than `n` (including the null character) that is stored in the character array `s`, and returns the number of characters (without the null character) that would have been stored if `n` were sufficiently large. The argument `n` shall be less than **SIZE_MAX** and `s` shall refer to a mutable byte array that has at least `n` elements. The textual conversion is similar to **snprintf** (with some exceptions) and is deduced from $A^{[x]}$, A, A^x and `baseflags`, if provided, as indicated below. Under any circumstances, no byte in `s` beyond the n^{th} is accessed and if `s` is modified it is null terminated within the first `n` bytes. The **locale** feature may be accessed to determine a multi-byte encoding if a wide-character or wide-string conversion is performed and if such a conversion fails, **errno** is set to **EILSEQ**; otherwise they are never accessed.

8 If $n \equiv 0$, the **totext** type-generic macro does not access `s` and returns the same number as above, and may therefore be used to prepare a suitably sized buffer to store the conversion. If $n > 0$ and `s` is a null pointer, **SIZE_MAX** is returned and no other action is performed. If $n \equiv 1$ (or < 5 for complex types) only a null character is stored in `s[0]` and the return value is as if `n` had been 0.

9 If $A^{[x]}$ is an array type, the elements of type A^x are printed as by the rules below, unless it is already considered to be text, that is, one of the following holds:

— The base is 0 and the array base type is a qualified or unqualified version of **char**.

— It is a qualified or unqualified version of **char8_t**, **wchar_t**, **char16_t** or **char32_t** and that type is a proper type.

— **totext_char** is set.

Then, in addition to the above constraints, `x` shall be a string or wide-string or an array of pointers to strings or wide-strings, and these strings or wide-strings shall start and end in the initial shift state. The strings or wide-strings are then stored in order in the buffer as decribed below.

10 If `x` is printed as an array (or array of strings or wide-strings) and if `glue` is provided, it is copied between each pair of consecutive elements of the array that are stored. If `glue` is omitted or a null pointer, a default of `"\t"` is used, unless the representation is

— a string representation or

— the representation of unsigned values, a precision is set and **totext_alt** is not set,

and in which case `""` is used.[398] Otherwise, `glue` is ignored.

11 Otherwise, if A is a pointer type or **nullptr_t**, it is converted to **void*** represented as with a `"%p"` specification, unless it is considered to be pointing to a string or wide-string, analogously determined as for arrays above, and if the value is a valid address the corresponding storage instance is exposed.In that case and if no precision is provided, `x` shall point to the first element of a string or wide-string which shall start and end in the initial shift state. If a precision is provided, the pointed-to character or wide-character array shall have at least as many elements. That string or wide-string is only accessed for reading and will never be accessed beyond the first null character or

[396] Note that this constraint exempts storage of textual information of arguments that are already considered to be text (such as strings or wide-strings) from some of the rule for the `core::invariant` and `core::state_conserving` attributes, namely the fact of whether or not the **totext** type-generic macro accesses lvalues of character type.

[397] Because the types $A^{[x]}$, A, and A^x as well as an optional `baseflags` argument are observable at translation time, a violation of these constraint can be diagnosed.

[398] That is strings (including **bool** represented as strings) and fixed precision integers without a sign are glued directly together, whereas all other types are separated with a tabulator character.

beyond the precision. Within that range it shall not overlap with s and a **core::noalias** attribute for x is implicitly assumed.

12 If A is a pointer to a character type including **char8_t**, a call with n > 0 copies at most n-1 or prec bytes, whichever is smaller, and terminates the target string such written with a null character. If no precision is provided, the return value is the position of the first null character in x. Otherwise, it is the minimum of that number and the precision.

13 If A is a pointer type to **wchar_t**, **char16_t** or **char32_t** the wide-characters are converted one-by-one as by **wcrtomb**, **c16rtomb** or **c32rtomb**, respectively, as long as the sequence and the terminating null character is known to fit into s and as long as the precision, if provided, is not exceeded. If the underlying types for wide-characters coincide in any way, the interpretation as UTF–32 is prefered over UTF–16, which is prefered over the implementation and **locale** dependent encoding of **wchar_t**. If the target encoding is a variable length encoding, the resulting multi-byte sequence for each source character (or, possibly, for a UTF–16 surrogate pair) is either completely stored or not stored at all. If an encoding error for one of the wide-characters occurs, the return value is **SIZE_MAX** and **errno** is set to **EILSEQ**; in that case it is unspecified how much of the conversion has been stored in s, if any. If no encoding error occurred, the return value is the length of the multi-byte string if the source wide-string would be completely converted within in the bound of the precision, if that is provided. The **locale** may be accessed to determine the source and target encoding and **errno** may be set in case of an encoding error, but otherwise no hidden state of the execution is accessed.

14 If A^x is a character type or wide-character type and considered to be text as specified above, the, possibly converted, character is stored if possible. If n > 1 and x is zero or a surrogate character, only a null character is stored in s[0], and the return value is 0. Otherwise, if A is a character type including **char8_t**, if n > 1, x is stored in s[0], followed by a null character in s[1] and the return value is 1. If x is a wide-character and the multi-byte conversion (including a terminating null character) does not fit into s, the length of the multi-byte string is returned and s is not accessed.

15 Otherwise, A^x is an arithmetic type and if n is larger than **INT_MAX** it is adjusted to that value. If A^x is a real type and base is not 2, a call with n > 1 is equivalent to a call to **snprintf** as follows, excepted that no **locale** category is accessed and that the conversion of floating point values is performed as if the **LC_NUMERIC** category were set to "C". If a precision had been specified and if for the chosen conversion specifier the **snprintf** function does define a behavior for a precision, the equivalence is to

```
(size_t)snprintf(s, n, fmt, (int)prec, x)
```

where fmt is chosen according to A^x and contains a .* precision specifier. Otherwise, the precision, if any, is ignored and the behavior is equivalent to

```
(size_t)snprintf(s, n, fmt, x)
```

with an appropriate fmt that does not contain a precision specification. In both cases, fmt contains length specifiers as needed for A^x and optional additional flags as specified below. If not specified otherwise, if there is a choice between a lowercase and an uppercase conversion specifier for **snprintf**, the lowercase variant is chosen. If **totext_up** is set, the upper case variant is chosen if it exists, otherwise this flag is ignored.

16 If A^x is a floating point type and the base is 0 or 10 the default conversion specifier for fmt is g or G, respectively; if **totext_exp** is specified it is e or E, and if **totext_fixed** is specified it is f or F. If the base is 16 the conversion specifier is a or A. Before the textual conversion, x is explicitly converted to its generic type.[399)]

17 If A^x is **bool** and **totext_char** is set, the effect is as storing the strings "false" or "true", respectively, as by an %s or %.*s conversion specifier. Otherwise, x is converted to **int** and the

[399)] The explicit conversion has the effect that if called with a floating point argument that has an evaluation format with a greater precision than its type, according to 5.2.4.2.2 the **totext** type-generic macro will convert such an argument to the precision of the argument type before storing the textual representation.

rules for that type apply.

18 If A^x is an integer type (and not considered as a character or wide-character) and the base is not 2, one of the integer conversion specifiers is used; if the base is 0 or 10 the specifiers are `d` or `u` depending on the signedness of A^x; if the base is 8 or 16 the specifiers are `o` or `x` (or `X`), respectively. If the base is 2, a binary representation with digits `0` and `1` is stored with analogous rules as for a `x` conversion. If by doing so a signed integer type would be converted with a format that corresponds to an unsigned type, `x` is first converted to the unsigned type.[400)]

19 If **`totext_alt`** is specified the rules for the **#** flag apply where possible, with the additional rule that for a base of 2 a prefix of `0b` or `0B` is used, respectively. If **`totext_sign`** is specified the rules for the **+** flag apply if the conversion may have a sign, that is `a`, `d`, `e`, `f`, or `g`. If they do not apply to the chosen conversion, both specifications are otherwise ignored

20 For `n` $\equiv 0$ the handling is similar, respectively, but without a buffer and without a specification of a field width:

```
  (size_t)snprintf(nullptr, 0, fmt, (int)prec, x)
  (size_t)snprintf(nullptr, 0, fmt, x)
```

21 If `x` has a complex type and `n` < 5 it is set to 0, and no conversion is performed and `s` is not accessed. Otherwise, the rules are similar to converting real and imaginary part in sequence. A call with a precision and **`totext_sign`** is equivalent to

```
(size_t)snprintf(s, n, fmt fmt "i", (int)prec, real_value(x),
                 (int)prec, imaginary_value(x))
```

where `fmt` is as above for the corresponding real floating type, and with `"I"` instead of `"i"` if **`totext_up`** is set. If no precision is specified an equivalent call without a precision specifier is executed analogously to the above. The effect is, that real and imaginary parts are converted as two real floating values, such that both have necessarily a sign and such that the imaginary part is followed by the character `i` or **`I`**.

22 If **`totext_sign`** is not set and neither the real nor the imaginary part are zero, the effect is similar:

```
(size_t)snprintf(s, n, fmt fmti "i", (int)prec, real_value(x),
                 (int)prec, imaginary_value(x))
```

only that the format `fmti` for the imaginary part has an additional **+** flag to force a sign between real and imaginary part. Otherwise, if the imaginary part is equal to zero, it is omitted, including the sign and the `i` or **`I`**, that is as if the call had been

```
totext(n, s, real_value(x), baseflags)
```

Otherwise, if the real part is zero, the imaginary part is converted similarly with the character `i` or **`I`** appended, that is the call is equivalent to a call, respectively,

```
totext(n-1, s, imaginary_value(x), baseflags)
```

followed by a call **`strcat`**`(s, "i")` or **`strcat`**`(s, "I")`.

Returns

23 The **`totext`** type-generic macro returns the number of characters that would have been stored had `n` been sufficiently large, not counting the terminating null character, or **`SIZE_MAX`** if `s` had been a null pointer where it shouldn't or if an encoding error occured. Thus, the null-terminated output has been completely stored if and only if the returned value is less than `n`. If an encoding error occured **`errno`** is set to **`EILSEQ`**.

[400)] The effect is that the only relevant information for converting integers with the **`totext`** type-generic macro is the bit representation of the integer type.

24 **NOTE 1** The intent is that the **totext** type-generic macro has defined behavior whenever it is presented with arguments that fulfill the above constraints and such that

— if A is a pointer to character or wide-character type and considered to be string or wide-string, the pointed-to array is accessible up to the next null character or to the precision, whicheve comes first; that string or wide-string does not overlap with s; an access to the **locale** does not race with any call to **setlocale**;

— if A is an object pointer type, x is not indeterminate;

— if $A^{[x]}$ is an array of one of the types above, these conditions hold for each element of the array;

— if provided, n is strictly less than **SIZE_MAX** and s specifies a modifiable byte array of size at least n.

Put aside different organization of the type system for wide-character types, the accepted formats are the same for all imlementations and all possible format errors are diagnosed by the translator.

25 **NOTE 2** The **totext** type-generic macro overcomes the restriction of **printf** family of functions concerning the their return value. This makes a difference for textual conversions, where **printf** would be restricted to strings that are shorter than **INT_MAX**, but where the **totext** type-generic macro can handle strings or wide-strings up to **SIZE_MAX** target characters. **printf** has that restriction, because otherwise the return value of the function can not be properly encoded. Here, for the **totext** type-generic macro this restriction still holds for the textual conversion of arithmetic or pointer types, but a conversion of such a value that exceeds this length is not very sensible.

26 **NOTE 3** The constraints allow to convert signed integer values with format specifiers for unsigned types, for example it allows to use a base of 16 for signed types. Since the underlying call to **snprintf** only uses the representation of the value and all representations for a signed type are valid for the corresponding unsigned type, such a specification is well-defined, but may have surprising results.

27 **NOTE 4** The choices for strings are such that per default only **char** (and perhaps **char8_t**) pointers are reliably interpreted as strings and all other data pointer types, even to wide-characters, may just store the textual representation of the pointer value. The textual representation of a pointer to **char** can be stored by converting the pointer to **void***. Wide character strings can be forceably converted to a multi-byte string by using **totext_char**. Thus, a call **totext**(n, s, str, **totext_char**) can be used to store str as multi-byte string regardless if str is a string or wide-string, and regardless which type is used for wide-strings.

28 **NOTE 5** Similarly, only **char** values are reliably interpreted as a character encodings. All other integer values, even if they are integer character constants (which may have type **int** or **char8_t**) or are wide-character constants, may be converted to the decimal representation per default. Interpretation as wide or plain characters may be forced by using **totext_char**. Then, a call **totext**(n, s, chr, **totext_char**) can be used to represent chr as a multi-byte string regardless if chr is a plain character, an integer character constant or a wide character.

29 **EXAMPLE** The following shows examples of the use of the **totext** type-generic macro :

```
size_t const mlen = totext(0, nullptr, ULLONG_MAX, "o") + 2; // + sign or prefix
char buffer[mlen+1];
// A[x], A, and A^x are all unsigned
totext(mlen, buffer, 7u, 8 U totext_alt); // stores the characters '0' and '7'
totext(mlen, buffer, LLONG_MIN);          // decimal, including the - sign

totext(mlen, buffer, 'A');                // depends on type of character literals
totext(mlen, buffer, L'A');               // depends on the wchar_t type

totext(mlen, buffer, (char)'A');          // stores the character 'A'
totext(mlen, buffer, 'A', totext_char);   // same
totext(mlen, buffer, L'A', totext_char);  // same, also for extended characters

totext(mlen, buffer, 'A', 10);            // stores the encoding of A in decimal
totext(mlen, buffer, L'A', 10);           // same, also for extended characters

static unsigned flux = 16 U totext_precision(8);
totext(mlen, buffer, ULLONG_MAX, flux);          // error, diagnosis ``is not an ICE''
inline const unsigned hex8 = 16 U totext_precision(8);
totext(mlen, buffer, ULLONG_MAX, hex8);          // ok

uint32_t a[] = { 300762161, 965766188, };
totext(mlen, buffer, a[0], hex8);         // stores "11ed4431"
// A[x] is uint32_t[2], A is uint32_t*, and A^x is uint32_t
totext(mlen, buffer, a, hex8);            // stores "11ed443139906c2c"
totext(mlen, buffer, a, 16 U totext_up); // stores "11ED4431\t39906C2C"

char const* b[] = { "this", "is", "a", "text", };
```

```
totext(mlen, buffer, b[0]);                // stores "this"

// A^[x] is char const*[4], A is char const**, and A^x is char const*
totext(mlen, buffer, b);                   // stores "thisisatext"
totext(mlen, buffer, b, 0, " ");           // stores "this is a text"

// uses locale to determine the multi-byte encoding
// A^[x] is const wchar_t[5], A is const wchar_t*, and A^x is wchar_t
totext(mlen, buffer, L"φρήν", totext_char); // stores a multi-byte sequence

// does not use locale
enum { hex4 = 16 U totext_precision(4), };
// A^[x] is const char16_t[5], A is const char16_t*, and A^x is char16_t
totext(mlen, buffer, u"φρήν", hex4);        // stores "03c603c103ae03bd0000"
totext(mlen, buffer, u"φρήν", hex4, "\\u"); // stores "03c6\u03c1\u03ae\u03bd\u0000"
```

7.22.3 Pseudo-random sequence generation functions

7.22.3.1 The rand function

Synopsis

1
```
#include <stdlib.h>
~~int rand(void);~~
int rand(void) [[core::modifies(rand)]];
```

Description

2 The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX** inclusive.

3 The **rand** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **rand** function.

Recommended practice

4 There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Returns

5 The **rand** function returns a pseudo-random integer.

Environmental limits

6 The value of the **RAND_MAX** macro shall be at least 32767.

7.22.3.2 The srand function

Synopsis

1
```
#include <stdlib.h>
~~void srand(unsigned int seed);~~
void srand(unsigned int seed) [[core::modifies(rand)]];
```

Description

2 The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence shall be generated as when **srand** is first called with a seed value of 1.

3 The **srand** function is not required to avoid data races with other calls to pseudo-random sequence

generation functions. The implementation shall behave as if no library function calls the `srand` function.

Returns

4 The `srand` function returns no value.

5 **EXAMPLE** The following functions define a portable implementation of `rand` and `srand`.

```
static unsigned long int next = 1;

int rand(void)   //  RAND_MAX assumed to be 32767
{
        next = next * 1103515245 + 12345;
        next = next × 1103515245 + 12345;
        return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
        next = seed;
}
```

7.22.4 Storage management functions

1 ~~The order and contiguity of storage allocated~~ If the allocation succeeds, the pointer to a storage instance returned by a call to ~~by successive calls to the~~ **`aligned_alloc`**, **`calloc`**, **`malloc`**, ~~and~~ or **`realloc`** ~~functions is unspecified. The pointer returned if the allocation succeeds~~ with size argument `size` (and possibly `nmemb`) points to the initial byte of an array of type **`void`**`[size]` or **`void`**`[size×nmemb]`, respectively, that is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested. It may then be used to access such an object or an array of such objects in the ~~space~~ storage instance allocated (until the ~~space~~ storage instance is explicitly deallocated). The lifetime of an allocated ~~object~~ storage instance extends from the allocation until the deallocation. Each such allocation shall yield a pointer to ~~an object~~ a storage instance that is disjoint from any other ~~object~~storage instance. The pointer returned points to the start ~~(lowest byte address)~~ address of the allocated ~~space~~storage instance. If the ~~space~~ storage instance cannot be allocated, a null pointer is returned. If the size of the ~~space~~ storage instance requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the ~~behavior is as if the size were some nonzero value, except that~~ address of a storage instance of size zero is returned. For the latter, the returned pointer shall not be used to access an object.

2 For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only ~~memory locations~~ storage instances accessible through their arguments and not other static duration storage ~~.~~ instances. These functions may, however, visibly modify the storage instance that they allocate or deallocate. Calls to these functions that allocate or deallocate storage instances in a particular region of ~~memory~~ the address space shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.[401)]

3 Storage management functions act as if they access a hidden state **`malloc`** which represents thread-specific state for bookkeeping.

Recommended practice

4 It is recommended that all implementation specific function declarations that deal with notions of storage allocation similar to this clause are annotated with the core function attribute **`core::modifies(malloc)`**.

7.22.4.1 The `aligned_alloc` function

401) This means that an implementation may only reuse a valid address that is computed from an allocated storage instance for a different allocated storage instance if the calls to allocate and deallocate the storage instances synchronize.

Synopsis

1
```
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
constexpr void * aligned_alloc(size_t alignment, size_t size)
    [[core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak]];
```

Description

2 The **aligned_alloc** function allocates ~~space for an object~~ a storage instance whose alignment is specified by `alignment`, whose size is specified by `size`, and whose ~~value is indeterminate~~byte values are unspecified. If the value of `alignment` is not a valid alignment supported by the implementation the function shall fail by returning a null pointer.

Returns

3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.4.2 The calloc function

Synopsis

1
```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
constexpr void * calloc(size_t nmemb, size_t size)
        [[core::noalias(nmemb × size), core::modifies(malloc), core::noleak]];
```

Description

2 The **calloc** function allocates ~~space~~ a storage instance for an array of `nmemb` objects, each of whose size is `size`. The ~~space~~ storage instance is initialized to all bits zero.[402]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.4.3 The free function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
constexpr void free(void * [[core::free]] ptr)
        [[core::modifies(malloc), core::noleak]];
```

Description

2 The **free** function causes the ~~space~~ storage instance pointed to by `ptr` to be deallocated, that is, made available for further ~~allocation.~~use.[403] If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.22.4.4 The malloc function

[402] Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

[403] That means that the implementation may reuse the address range of the storage instance (determined by `ptr` and its size) for any storage instance whose instantiation synchronizes with the call.

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
constexpr void * malloc(size_t size)
  [[core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak]];
```

Description

2 The **malloc** function allocates ~~space for an object~~ a storage instance whose size is specified by `size` and whose ~~value is indeterminate~~byte values are unspecified.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.4.5 The `realloc` function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
constexpr void * realloc(void * [[core::realloc]] ptr, size_t size)
  [[core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak]];
```

Description

2 The **realloc** function deallocates the old ~~object~~ storage instance pointed to by `ptr` and returns a pointer to a new ~~object~~ storage instance that has the size specified by `size`. The ~~contents of the new object shall be the same as that~~ bytes of the old ~~object prior to deallocation,~~ storage instance up to the lesser of the new and old sizes ~~.~~ are copied as if by **memcpy** to the initial bytes of the new storage instance. Any bytes in the new ~~object~~ storage instance beyond the size of the old object have ~~indeterminate~~ unspecified values.

3 If `ptr` is a null pointer, the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If `size` is nonzero and ~~memory for the new object is not~~ no storage instance is allocated, the old ~~object~~ storage instance is not deallocated. If `size` is zero and ~~memory for the new object is not~~ no storage instance is allocated, it is implementation-defined whether the old ~~object~~ storage instance is deallocated. If the old ~~object~~ storage instance is not deallocated, ~~its value~~ it shall be unchanged.

Returns

4 The **realloc** function returns a pointer to the new ~~object~~ storage instance (which may have the same value as a pointer to the old ~~object),~~ storage instance), or a null pointer if ~~the new object has not~~ no new storage instance has been allocated.

5 NOTE If a call to **realloc** is successful, the initial part of the new storage instance represents objects with same value and effective type as the initial part of the old storage instance, if any. Nevertheless, the new storage instance has to be considered to be different from the old one:

— Even if both storage instances have the same address, all pointers to the old storage instance (stored within or outside the storage instance) are invalid because that storage instance ceases to exist.

— Copies of opaque objects in the new storage instance may need explicit initialization or otherwise be in an indeterminate state.

— Resources reserved for opaque objects in the old storage instance that have hidden state and need destruction (such as variable argument lists, mutexes or condition variables) may be squandered.

7.22.5 Communication with the environment

7.22.5.1 The `abort` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void abort(void);
_Noreturn void abort(void)
  [[core::reentrant, core::modifies(fopen), core::evaluates(stdin, stdout, stderr)]];
```

Description

2 The **abort** function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise**(**SIGABRT**).

Returns

3 The **abort** function does not return to its caller.

7.22.5.2 The atexit function

Synopsis

1
```
#include <stdlib.h>
int atexit(void (*func)(void));
int atexit(void (*func)(void)) [[core::modifies(atexit)]];
```

Description

2 The **atexit** function registers the function pointed to by `func`, called the ***atexit*** *handler*, to be called without arguments at normal program termination.[404] It is unspecified whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The **atexit** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **at_quick_exit** function (7.22.5.3), the **exit** function (7.22.5.4).

7.22.5.3 The at_quick_exit function

Synopsis

1
```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
int at_quick_exit(void (*func)(void)) [[core::modifies(at_quick_exit)]];
```

Description

2 The **at_quick_exit** function registers the function pointed to by `func`, called the ***at_quick_exit*** *handler*, to be called without arguments should **quick_exit** be called.[405] It is unspecified whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

[404] The **atexit** function registrations are distinct from the **at_quick_exit** registrations, so applications might need to call both registration functions with the same argument.

[405] The **at_quick_exit** function registrations are distinct from the **atexit** registrations, so applications might need to call both registration functions with the same argument.

Returns

4 The **`at_quick_exit`** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **`quick_exit`** function (7.22.5.7).

7.22.5.4 The `exit` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void exit(int status);
_Noreturn void exit(int status)
        [[core::modifies(atexit, fopen), core::evaluates(stdin, stdout, stderr)]];
```

Description

2 The **`exit`** function causes normal program termination to occur. No ~~functions registered by the~~ **`at_quick_exit`** ~~function~~ handlers are called. If a program calls the **`exit`** function more than once, or calls the **`quick_exit`** function in addition to the **`exit`** function, the behavior is undefined.

3 First, all ~~functions registered by the~~ **`atexit`** ~~function~~ handlers are called, in the reverse order of their registration,[406] except that a ~~function~~ handler is called after any previously registered ~~functions~~ handlers that had already been called at the time it was registered. If, during the call to any such ~~function~~handler, a call to the **`longjmp`** function is made that would terminate the call to the registered ~~function~~handler, the behavior is undefined. There is a sequence point before the first call to a registered **`atexit`** handler, if any, that synchronizes with the termination of all threads, as described for **`thrd_exit`** (7.26.5.5). Furthermore, there is a sequence point immediately before and immediately after each call to an **`atexit`** handler.

4 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **`tmpfile`** function are removed.

5 Finally, control is returned to the host environment. If the value of `status` is zero or **`EXIT_SUCCESS`**, an implementation-defined form of the status *successful termination* is returned. If the value of `status` is **`EXIT_FAILURE`**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

6 The **`exit`** function cannot return to its caller.

7.22.5.5 The `_Exit` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void _Exit(int status);
_Noreturn void _Exit(int status)

    [[core::reentrant, core::modifies(fopen), core::evaluates(stdin, stdout, stderr)]];
```

Description

2 The **`_Exit`** function causes normal program termination to occur and control to be returned to the host environment. No ~~functions registered by the~~ **`atexit`** ~~function, the~~ handlers, **`at_quick_exit`** ~~function~~handlers, or signal handlers registered by the **`signal`** function are called. The status returned to the host environment is determined in the same way as for the **`exit`** function (7.22.5.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

Returns

3 The **`_Exit`** function cannot return to its caller.

[406] Each ~~function~~ handler is called as many times as it was registered, and in the correct order with respect to other registered ~~functions~~handlers.

7.22.5.6 The `getenv` function

Synopsis

1
```
#include <stdlib.h>
char *getenv(const char *name);
char * [[ core::alias ]] getenv(const char *name) [[ core::evaluates(environ) ]] ;
```

Description

2 The **`getenv`** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation-defined. The **`getenv`** function need not avoid data races with other threads of execution that modify the environment list.[407)]

3 The implementation shall behave as if no library function calls the **`getenv`** function.

Returns

4 The **`getenv`** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **`getenv`** function. If the specified `name` cannot be found, a null pointer is returned.

7.22.5.7 The `quick_exit` function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void quick_exit(int status);
_Noreturn void quick_exit(int status)

[[ core::reentrant, core::modifies(at_quick_exit, fopen), core::evaluates(stdin, stdout, stderr) ]] ;
```

Description

2 The **`quick_exit`** function causes normal program termination to occur. No ~~functions registered by the~~ **`atexit`** ~~function~~ handlers or signal handlers registered by the **`signal`** function are called. If a program calls the **`quick_exit`** function more than once, or calls the **`exit`** function in addition to the **`quick_exit`** function, the behavior is undefined. If a signal is raised while the **`quick_exit`** function is executing, the behavior is undefined.

3 ~~The first calls all functions registered by the~~ First, all **`at_quick_exit`** ~~function~~handlers are called, in the reverse order of their registration,[408)] except that a ~~function~~ handler is called after any previously registered ~~functions~~ handlers that had already been called at the time it was registered. If, during the call to any such ~~function~~handler, a call to the **`longjmp`** function is made that would terminate the call to the registered ~~function~~handler, the behavior is undefined. There is a sequence point before the first call to a registered **`at_quick_exit`** handler, if any, that synchronizes with the termination of all threads, as described for **`thrd_exit`** (7.26.5.5). Furthermore, there is a sequence point immediately before and immediately after each call to an **`at_quick_exit`** handler.

4 Then control is returned to the host environment by means of the function call **`_Exit`**`(status)`.

Returns

5 The **`quick_exit`** function cannot return to its caller.

7.22.5.8 The `system` function

Synopsis

1
```
#include <stdlib.h>
int system(const char *string);
int system(const char *string) [[ core::modifies(fopen, time) ]] ;
```

[407)] Many implementations provide non-standard functions that modify the environment list.

[408)] Each ~~function~~ handler is called as many times as it was registered, and in the correct order with respect to other registered ~~functions~~handlers.

Description

2 If `string` is a null pointer, the `system` function determines whether the host environment has a *command processor*. If `string` is not a null pointer, the `system` function passes the string pointed to by `string` to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling `system` to behave in a non-conforming manner or to terminate.

Returns

3 If the argument is a null pointer, the `system` function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the `system` function does return, it returns an implementation-defined value.

7.22.6 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as `size_t nmemb` specifies the length of the array for a function, `nmemb` can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. Pointer arguments on such a call shall still have valid values, as described in 7.1.4.

2 The implementation shall ensure that the second argument of the comparison function (when called from `bsearch`), or both arguments (when called from `qsort`), are pointers to elements of the array.[409] The first argument when called from `bsearch` shall equal `key`.

3 The comparison function shall not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

4 When the same objects (consisting of `size` bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for `qsort` they shall define a total ordering on the array, and for `bsearch` the same object shall always compare the same way with the key.

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

7.22.6.1 The `bsearch` type-generic macro

Synopsis

1
```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
C *bsearch(const void *key, C *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *))
        [[core::alias(base)]];
```

Constraints

2 The second argument to a call shall have pointer to object type that is not `volatile` qualified.

Description

3 The `bsearch` type-generic macro searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element of the array is specified by `size`.

[409] That is, if the value passed is `p`, then the following expressions are always nonzero:

```
((char *)p - (char *)base) % size ≡ 0
(char *)p ≥ (char *)base
(char *)p  <  (char *)base + nmemb × size
```

4 The type `C` is **`void`** or **`const void`**. A call to the **`bsearch`** type-generic macro first converts the second argument to the version of **`void`** that has the same **`const`**-qualification as the pointed-to type of that argument.

5 The comparison function pointed to by `compar` is called with two arguments that point to the `key` object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.[410]

Returns

6 The **`bsearch`** type-generic macro returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

7 **NOTE** The C and C++ standards have a function of the same name that returns a pointer to the second argument that drops a **`const`**-qualifier from the pointed-to type. Applications should prefer to use the type-generic macros to avoid write-privilege escalation on **`const`** qualified arrays.

7.22.6.2 The `qsort` function

Synopsis

1
```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description

2 The **`qsort`** function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.

3 The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

Returns

5 The **`qsort`** function returns no value.

7.22.7 Integer arithmetic functions

7.22.7.1 The `abs`, `labs`, and `llabs` functions

Synopsis

1
```
#include <stdlib.h>
        int abs(int j);
        long int labs(long int j);
        long long int llabs(long long int j);
[[deprecated("use abs type-generic macro")]] int abs(int j);
[[deprecated("use abs type-generic macro")]] long int labs(long int j);
[[deprecated("use abs type-generic macro")]] long long int llabs(long long int j);
```

Description

2 The **`abs`**~~, `labs`, and `llabs` functions~~ obsolescent functions **`abs`**, **`labs`**, and **`llabs`** compute the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[411]

[410] In practice, the entire array is sorted according to the comparison function.
[411] The absolute value of the most negative number is not representable.

Returns

3 The ~~abs, labs, and llabs~~ **abs**, **labs**, and **llabs**, functions return the absolute value.

Recommended practice

4 Because the **abs** type-generic macro from the `<math.h>` header is able to represent all result values in its return type, it is recommended that applications prefer that macro over the functions described here.

7.22.7.2 The div type-generic macro

Synopsis

1
```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
constexpr Q div(R numer, S denom);
```

Constraints

2 The argument types *R* and *S* shall be integer types.

Description

3 The ~~div, ldiv, and lldiv, functions compute~~ **div** type-generic macro computes `numer/denom` and `numer%denom` in a single operation.

4 The argument values shall be such that there is a signed type that can hold the value. For *R* and *S* the types of the arguments, let *Q* be the signed integer type of minum rank that can hold the values of *R* and *S*, if any, or **long long int** if there is no such signed type. The argument values are converted to *Q*.

5 Outside a function call, the **div** type-generic macro can be converted to a function pointer type `auto(*)(Q, Q)` where `Q` is a wide signed integer type.

Returns

6 The ~~div, ldiv, and lldiv functions return a structure of type div_t, ldiv_t, and lldiv_t, respectively,~~ **div** type-generic macro returns a complete structure type comprising both the quotient and the remainder. The ~~structures~~ structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has the ~~same type as the arguments numer and denom~~ type *Q*. If either part of the result cannot be represented, the behavior is undefined.

7 **EXAMPLE**

```
auto res_int = div(127, 13);
printf("result is %d, %d\n", res_int.quot, res_int.rem);
...
auto (*funcp)(long, long) = div;
auto res_long = funcp(127, 13);
printf("result is %ld, %ld\n", res_long.quot, res_long.rem);
```

For `res_int` the type-generic macro is directly called with two **int** values. Therefore the result has the two return values as **int**. For `funcp`, the macro is converted to a function pointer, which is then used in a call producing the result `res_long`. Since the prototype for the function pointer uses **long**, the two **int** values are first converted to **long** and the two result values are then of type **long**, too.

7.22.8 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each of the **mbtowc** and **wctomb** functions is placed into its initial conversion state at program startup and can be returned to that state by a call for which its character pointer argument, `s`, is a null pointer. Subsequent calls with `s` as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with `s` as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and

zero otherwise.[412] Changing the **LC_CTYPE** category causes the conversion state of the **mbtowc** and **wctomb** functions to be indeterminate.

7.22.8.1 The `mblen` function

Synopsis

1

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
int mblen(const char *s, size_t n)
      [[core::evaluates(locale)]];
```

Description

2 If `s` is not a null pointer, the **mblen** function determines the number of bytes contained in the multibyte character pointed to by `s`. Except that the conversion state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, (const char *)0, 0);
mbtowc((wchar_t *)0, s, n);
```

Returns

3 If `s` is a null pointer, the **mblen** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If `s` is not a null pointer, the **mblen** function either returns 0 (if `s` points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next `n` or fewer bytes form a valid multibyte character), or returns `-1` (if they do not form a valid multibyte character).

Forward references: the **mbtowc** function (7.22.8.2).

7.22.8.2 The `mbtowc` function

Synopsis

1

```
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);

int mbtowc(wchar_t * [[core::noalias]] pwc, const char * [[core::noalias]] s, size_t n)
      [[core::evaluates(locale)]];
```

Description

2 If `s` is not a null pointer, the **mbtowc** function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide character is the null wide character, the function is left in the initial conversion state.

3 The implementation shall behave as if no library function calls the **mbtowc** function.

Returns

4 If `s` is a null pointer, the **mbtowc** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If `s` is not a null pointer, the **mbtowc** function either returns 0 (if `s` points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next `n` or fewer bytes form a valid multibyte character), or returns `-1` (if they do not form a valid multibyte character).

5 In no case will the value returned be greater than `n` or the value of the **MB_CUR_MAX** macro.

7.22.8.3 The `wctomb` function

[412] If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

Synopsis

1
```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
int wctomb(char *s, wchar_t wc)
        [[core::evaluates(locale)]];
```

Description

2 The **wctomb** function determines the number of bytes needed to represent the multibyte character corresponding to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s` (if `s` is not a null pointer). At most **MB_CUR_MAX** characters are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

3 The implementation shall behave as if no library function calls the **wctomb** function.

Returns

4 If `s` is a null pointer, the **wctomb** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If `s` is not a null pointer, the **wctomb** function returns `-1` if the value of `wc` does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of `wc`.

5 In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

7.22.9 Multibyte/wide string conversion functions

1 The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

7.22.9.1 The mbstowcs function

Synopsis

1
```
#include <stdlib.h>
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);

size_t mbstowcs(wchar_t *[[core::noalias]] pwcs, const char *[[core::noalias]] s, size_t n)
        [[core::evaluates(locale)]];
```

Description

2 The **mbstowcs** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by `s` into a sequence of corresponding wide characters and stores not more than `n` wide characters into the array pointed to by `pwcs`. No multibyte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multibyte character is converted as if by a call to the **mbtowc** function, except that the conversion state of the **mbtowc** function is not affected.

3 No more than `n` elements will be modified in the array pointed to by `pwcs`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If an invalid multibyte character is encountered, the **mbstowcs** function returns `(size_t)(-1)`. Otherwise, the **mbstowcs** function returns the number of array elements modified, not including a terminating null wide character, if any.[413)]

7.22.9.2 The wcstombs function

[413)]The array will not be null-terminated if the value returned is `n`.

Synopsis

1
```
#include <stdlib.h>
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);

size_t wcstombs(char * [[core::noalias]] s, const wchar_t * [[core::noalias]] pwcs, size_t n)
            [[core::evaluates(locale)]];
```

Description

2 The **wcstombs** function converts a sequence of wide characters from the array pointed to by pwcs into a sequence of corresponding multibyte characters that begins in the initial shift state, and stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each wide character is converted as if by a call to the **wctomb** function, except that the conversion state of the **wctomb** function is not affected.

3 No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If a wide character is encountered that does not correspond to a valid multibyte character, the **wcstombs** function returns (**size_t**)(-1). Otherwise, the **wcstombs** function returns the number of bytes modified, not including a terminating null character, if any.[413)]

7.23 `_Noreturn` `<stdnoreturn.h>`

1 The header `<stdnoreturn.h>` defines the macro

```
noreturn
```

which expands to **`_Noreturn`**.

7.24 String and storage handling <string.h>

1 ~~The header~~ The <string.h> ~~declares one type and several functions , and defines one macro useful for manipulating arrays of character type~~ header presents functions and ~~other objects treated as arrays of character type.~~ macros that operate on storage on a byte level or on the level of wide-characters. Functions or macros with names starting with **str** receive byte arrays that are assumed to be of character or wide-character type; functions or macros with names starting with **mem** make no such assumptions and are suited to operate on the representation of any object type.

2 Functions and macros in this clause do not modify any bytes to which their arguments point unless they are described as modifying them. ~~The type is **size_t** and the macro is **NULL** (both described in 7.19). Various methods are used for determining the lengths of the arrays, but in all cases a **char** * or **void** * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.~~ If a pointed-to type of any of their arguments is **volatile** qualified the corresponding rules for **volatile** access apply. The synopsis give descriptions of type-generic macros in terms of supported prototypes. Within these prototypes the type of which they depend are designated with C or D and constraints to the possible combinations of these types apply as indicated. Calls to these macros shall then have arguments that are consistent with one of the possible prototypes.

3 The feature test macro **__CORE_VERSION_STRING_H__** expands to the token *202005*L.

4 **NOTE** For each type-generic macro, the C and C++ standards have a function of the same name. Unfortunately, many of them return a value that drops a **const**-qualifier from the pointed-to type of a parameter. Also, the functions can only be called for **volatile** qualified objects, if the qualification is cast away, and thus the load operations that are performed will generally not be conforming to the requirements for **volatile** objects. Applications should prefer to use the type-generic macros to avoid write-privilege escalation on **const** qualified byte arrays.

7.24.1 Conventions

7.24.1.1 String function conventions

1 The header <string.h> declares several functions and macros starting with the prefix **str** that are useful for manipulating arrays of character or wide-character type that are presented to the functions or macros as parameters of type pointer to a qualified or unqualified character type.[414] If for a type-generic macro parameter types are indicated as pointers to types C or D, these are qualified or unqualified character types or **wchar_t**, where D has the same generic type as has C. Various methods are used for determining the lengths of the arrays, but in all cases a (possibly qualified) pointer to character or wide-character argument points to the initial (lowest addressed) character or wide-character of the array. If an array is accessed beyond the end of its storage instance, or if the effective type of the object is not a character or wide-character array of the indicated type the behavior is undefined.

2 Where an argument is declared as **size_t** n that specifies the length, the array shall have at least n characters or wide-characters. Unless explicitly stated otherwise, if n has the value zero on a call, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a character or wide-characters finds no occurrence, a function that compares two character or wide-character sequences returns zero, and a function that copies characters or wide-characters copies none.

3 Where no argument **size_t** n declares the length of the array~~for a function ,~~, a corresponding array is assumend to be a string or wide-string. Any array passed to such a function through a (possibly qualified) pointer to character or wide-character argument shall be null terminated.

4 For all these functions, each character (that is not accessed as wide-character) shall be interpreted as if it had the type **unsigned char**.

7.24.1.2 Storage function conventions

1 The header <string.h> also declares several functions with a **mem** prefix that are useful for manipulating bytes of storage instances seen as an array of qualified or unqualified version of **void** and that are presented to the functions through parameters of type pointer to a qualified or unqualified

[414] See "future library directions" (7.32.12).

version of **void**,[415] and if for a type-generic macro parameter types are indicated as pointers to types C or D, these are qualified or unqualified versions of **void**. If a byte array is accessed beyond the end of its storage instance, the behavior is undefined.

2 Arguments declared as **size_t** n specify the length of the byte array; n can have the value zero on a call to that function. Unless explicitly stated otherwise~~in the description of a particular function in this subclause~~, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a ~~character~~ byte finds no occurrence, a function that compares two ~~character~~ byte sequences returns zero, and a function that copies ~~characters copies zero characters~~bytes copies none.

3 For all ~~functions in this subclause, each character~~ these functions, each byte shall be interpreted as if it had the type **unsigned char** (and therefore every possible object representation is valid and has a different value).

7.24.2 Copying functions

1 If the object representation of a non-null pointer is copied by a copying function, either directly or within an aggregate or union object, the pointer copy has the same provenance as the original.

2 If the object representation of an opaque object that is not an array of type **void** is copied by a copying function, either directly or within an aggregate or union object, the values of the representation bytes are copied into the target array, but the copy does not acquire a valid state for the opaque object type. If in that case in the calling context the target byte array previously had no effective type, henceforth it has the opaque object type as effective type.

7.24.2.1 The memcpy type-generic macro

Synopsis

1
```
#include <string.h>
~~void *memcpy(void * restrict s1, const void * restrict s2, size_t n);~~

C *memcpy(C * [[ core::noalias, core::writethrough ]] s1, D * [[ core::noalias ]] s2, size_t n)
        [[ core::alias(s1) ]] ;
```

Constraints

2 The first argument to a call shall be a pointer to object type that is not **const** qualified.

Description

3 The **memcpy** type-generic macro copies n ~~characters from the object~~ bytes from the byte array pointed to by s2 into the ~~object~~ byte array pointed to by s1. If copying takes place between ~~objects~~ byte arrays that overlap, the behavior is undefined.

4 Before the call, the first and second argument are converted to a **void** pointer as if by **tovoidptr**.

Returns

5 The **memcpy** type-generic macro returns the converted value of s1.

7.24.2.2 The memccpy type-generic macro

Synopsis

1
```
#include <string.h>
~~void *memccpy(void * restrict s1, const void * restrict s2, int c, size_t n);~~

C *memccpy(C * [[ core::noalias, core::writethrough ]] s1, D * [[ core::noalias ]] s2, int c, size_t n)
        [[ core::alias(s1) ]] ;
```

Constraints

2 The first argument to a call shall be a pointer to object type that is not **const** qualified.

[415]See "future library directions" (7.32.12).

Description

3 The `memccpy` type-generic macro copies ~~characters from the object~~ bytes from the byte array pointed to by `s2` into the ~~object~~ byte array pointed to by `s1`, stopping after the first occurrence of ~~character~~ byte `c` (converted to an `unsigned char`) is copied, or after `n` ~~characters~~ bytes are copied, whichever comes first. If copying takes place between ~~objects~~ byte arrays that overlap, the behavior is undefined.

4 Before the call, the first and second argument are converted to a `void` pointer as if by `tovoidptr`.

Returns

5 The `memccpy` type-generic macro returns a pointer to the ~~character~~ byte after the copy of `c` in `s1`, or a null pointer if `c` was not found in the first `n` ~~characters~~ bytes of `s2`.

7.24.2.3 The memmove type-generic macro

Synopsis

1
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
C *memmove(C *s1, D *s2, size_t n)
        [[core::alias(s1)]];
```

Constraints

2 The first argument to a call shall be a pointer to object type that is not `const` qualified.

Description

3 The `memmove` type-generic macro copies `n` ~~characters from the object~~ bytes from the byte array pointed to by `s2` into the ~~object~~ byte array pointed to by `s1`. Copying takes place as if the `n` ~~characters from the object~~ bytes from the byte array pointed to by `s2` are first copied into a temporary array of `n` ~~characters~~ bytes that does not overlap the ~~objects~~ byte arrays pointed to by `s1` and `s2`, and then the `n` ~~characters~~ bytes from the temporary array are copied into the ~~object~~ byte array pointed to by `s1`.

4 Before the call, the first and second argument are converted to a `void` pointer as if by `tovoidptr`.

Returns

5 The `memmove` type-generic macro returns the converted value of `s1`.

7.24.2.4 The strcpy type-generic macro

Synopsis

1
```
#include <string.h>
char *strcpy(char * restrict s1, const char * restrict s2);
C *strcpy(C * [[core::noalias, core::writethrough]] s1, D * [[core::noalias]] s2)
        [[core::alias(s1)]];
```

Constraints

2 The first argument to a call shall be a pointer to character type or `wchar_t` that is not `const` qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 The `strcpy` type-generic macro copies the string or wide-string, respectively, pointed to by `s2` (including the terminating null character or wide-character) into the array pointed to by `s1`. If copying takes place between ~~objects~~ strings or wide-strings that overlap, the behavior is undefined.

Returns

4 The `strcpy` type-generic macro returns the value of `s1`.

7.24.2.5 The strncpy type-generic macro

Synopsis

1
```
    #include <string.h>
    char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
  C *strncpy(C * [[core::noalias, core::writethrough]] s1, D * [[core::noalias]] s2, size_t n)
          [[core::alias(s1)]];
```

Constraints

2 The first argument to a call shall be a pointer to character type or **wchar_t** that is not **const** qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 The **strncpy** type-generic macro copies not more than `n` characters ~~(characters~~ or wide-characters (characters or wide-characters that follow a null character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.[416] If copying takes place between ~~objects~~ strings or wide-strings, respectively, that overlap, the behavior is undefined.

4 If the array pointed to by `s2` is a string or wide-string that is shorter than `n` characters or wide-characters, respectively, null characters or wide-characters are appended to the copy in the array pointed to by `s1`, until `n` characters or wide-characters in all have been written.

Returns

5 The **strncpy** type-generic macro returns the value of `s1`.

7.24.3 Concatenation functions

7.24.3.1 The strcat type-generic macro

Synopsis

1
```
    #include <string.h>
    char *strcat(char * restrict s1, const char * restrict s2);
    C *strcat(C * [[core::noalias]] s1, D * [[core::noalias]] s2)
          [[core::alias(s1)]];
```

Constraints

2 The first argument to a call shall be a pointer to character type or **wchar_t** that is not **const** qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 The **strcat** type-generic macro appends a copy of the string or wide-string, respectively, pointed to by `s2` (including the terminating null character or wide-character) to the end of the string or wide-string pointed to by `s1`. The initial character or wide-character of `s2` overwrites the null character or wide-character at the end of `s1`. If copying takes place between ~~objects~~ strings or wide-strings that overlap, the behavior is undefined.

Returns

4 The **strcat** type-generic macro returns the value of `s1`.

7.24.3.2 The strncat type-generic macro

Synopsis

1
```
    #include <string.h>
    char *strncat(char * restrict s1, const char * restrict s2, size_t n);
    C *strncat(C * [[core::noalias]] s1, D * [[core::noalias]] s2, size_t n)
          [[core::alias(s1)]];
```

[416] Thus, if there is no null character or wide-character in the first `n` characters or wide-character of the array pointed to by `s2`, the result will not be null-terminated.

Constraints

2 The first argument to a call shall be a pointer to character type or **wchar_t** that is not **const** qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 The **strncat** type-generic macro appends not more than n characters (a null character and characters that follow it are not appended) from the array pointed to by s2 to the end of the string or wide-string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character or wide-character is always appended to the result.[417] If copying takes place between ~~objects~~ strings or wide-strings that overlap, the behavior is undefined.

Returns

4 The **strncat** type-generic macro returns the value of s1.

Forward references: the **strlen** ~~function (??).~~ type-generic macro (7.24.6.4).

7.24.4 Comparison functions

1 The sign of a nonzero value returned by the comparison functions **memcmp**, **strcmp**, and **strncmp** is determined by the sign of the difference between the values of the first pair of ~~characters~~ bytes (both interpreted as **unsigned char**) that differ in the objects being compared.

7.24.4.1 The memcmp type-generic macro

Synopsis

1
```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
int memcmp(C *s1, D *s2, size_t n);
```

Description

2 The **memcmp** type-generic macro compares the first n ~~characters of the object~~ bytes of the byte array pointed to by s1 to the first n ~~characters of the object~~ bytes of the byte array pointed to by s2.[418]

3 Before the call, the first and second argument are converted to a **void** pointer as if by **tovoidptr**.

Returns

4 The **memcmp** type-generic macro returns an integer greater than, equal to, or less than zero, accordingly as the ~~object~~ byte array pointed to by s1 is greater than, equal to, or less than the ~~object~~ byte array pointed to by s2.

7.24.4.2 The strcmp type-generic macro

Synopsis

1
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
int strcmp(C *s1, D *s2);
```

Description

2 The **strcmp** type-generic macro compares the string or wide-string, respectively, pointed to by s1 to the string or wide-string pointed to by s2.

[417] Thus, the maximum number of characters or wide-characters that can end up in the array pointed to by s1 is **strlen**(s1)+n+1.

[418] The contents of "holes" used as padding for purposes of alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions can also cause problems in comparison.

Returns

3 The **strcmp** type-generic macro returns an integer greater than, equal to, or less than zero, accordingly as the string or wide-string pointed to by `s1` is greater than, equal to, or less than the string or wide-string pointed to by `s2`.

7.24.4.3 The `strcoll` type-generic macro

Synopsis

1
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
int strcoll(C *s1, D *s2)
      [[ core::evaluates(locale) ]];
```

Description

2 The **strcoll** type-generic macro compares the string or wide-string, respectively, pointed to by `s1` to the string or wide-string pointed to by `s2`, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Returns

3 The **strcoll** type-generic macro returns an integer greater than, equal to, or less than zero, accordingly as the string or wide-string pointed to by `s1` is greater than, equal to, or less than the string or wide-string pointed to by `s2` when both are interpreted as appropriate to the current locale.

7.24.4.4 The `strncmp` type-generic macro

Synopsis

1
```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
int strncmp(C *s1, D *s2, size_t n);
```

Description

2 The **strncmp** type-generic macro compares not more than `n` characters ~~(characters~~ or wide-characters (characters or wide-characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Returns

3 The **strncmp** type-generic macro returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

7.24.4.5 The `strxfrm` type-generic macro

Synopsis

1
```
#include <string.h>
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);

size_t strxfrm(C * [[ core::noalias, core::writethrough ]] s1, D * [[ core::noalias ]] s2, size_t n);
```

Constraints

2 The first argument to a call shall be a pointer to character type or **wchar_t** that is not **const** qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 The **strxfrm** type-generic macro transforms the string or wide-string pointed to by `s2` and places the resulting string or wide-string into the array pointed to by `s1`. The transformation is such that if the **strcmp** function is applied to two transformed strings or wide-strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the

same two original strings or wide-strings. No more than `n` characters or wide-characters are placed into the resulting array pointed to by `s1`, including the terminating null character or wide-character, respectively. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 The **`strxfrm`** type-generic macro returns the length of the transformed string or wide-string (not including the terminating null character or wide-character). If the value returned is `n` or more, the contents of the array pointed to by `s1` are indeterminate.

5 **EXAMPLE** The value of the following expression is the size of the array needed to hold the transformation of the string or wide-string pointed to by `s`.

```
        1 + strxfrm(NULL, s, 0)
        1 + strxfrm(nullptr, s, 0)
```

7.24.5 Search functions

7.24.5.1 The memchr type-generic macro

Synopsis

1
```
        #include <string.h>
        void *memchr(const void *s, int c, size_t n);
        C *memchr(C *s, int c, size_t n)
               [[ core::alias(s) ]];
```

Constraint

2 The first argument to the **`memchr`** type-generic macro shall be a pointer to object type.

Description

3 The **`memchr`** type-generic macro locates the first occurrence of `c` (converted to an **`unsigned char`**) in the initial `n` ~~characters~~ bytes (each interpreted as **`unsigned char`**) of the ~~object~~ byte array pointed to by `s`. The implementation shall behave as if it reads the ~~characters~~ bytes sequentially and stops as soon as a matching ~~character~~ byte is found.

4 The **`memchr`** type-generic macro behaves as if the first argument was first implicitly converted to a **`void`** pointer as if by **`tovoidptr`**.

Returns

5 The **`memchr`** type-generic macro returns a pointer to the located ~~character~~byte, or a null pointer if the ~~character~~ byte does not occur in the ~~object~~byte array.

7.24.5.2 The strchr type-generic macro

Synopsis

1
```
        #include <string.h>
        char *strchr(const char *s, int c);
        C *strchr(C *s, D c) [[ core::alias(s) ]];
```

Constraints

2 The first argument shall have pointer to character or wide-character type.

Description

3 The **`strchr`** type-generic macro locates the first occurrence of `c` (converted to ~~a **`char`**~~ `C`) in the string or wide-string pointed to by `s`. The terminating null character or wide-character is considered to be part of the string or wide-string.

4 The type `D` is the promoted generic type of `C`.

Returns

5 The **strchr** type-generic macro returns a pointer to the located character or wide-character, respectively, or a null pointer if the character or wide-character does not occur in the string or wide-string.

7.24.5.3 The `strcspn` type-generic macro

Synopsis

1
```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
size_t strcspn(C *s1, D *s2);
```

Description

2 The **strcspn** type-generic macro computes the length of the maximum initial segment of the string or wide-string pointed to by `s1` which consists entirely of characters or wide-characters *not* from the string or wide-string pointed to by `s2`.

Returns

3 The **strcspn** type-generic macro returns the length of the segment.

7.24.5.4 The `strpbrk` type-generic macro

Synopsis

1
```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
C *strpbrk(C *s1, D *s2)
        [[ core::alias(s1) ]];
```

Description

2 The **strpbrk** type-generic macro locates the first occurrence in the string or wide-string pointed to by `s1` of any character or wide-character, respectively, from the string or wide-string pointed to by `s2`.

Returns

3 The **strpbrk** type-generic macro returns a pointer to the character or wide-character, respectively, or a null pointer if no character or wide-character from `s2` occurs in `s1`.

7.24.5.5 The `strrchr` type-generic macro

Synopsis

1
```
#include <string.h>
char *strrchr(const char *s, int c);
C *strrchr(C *s, D c)
        [[ core::alias(s) ]];
```

Description

2 The **strrchr** type-generic macro locates the last occurrence of `c` (converted to a **char**) in the string or wide-string, respectively, pointed to by `s`. The terminating null character or wide-character, respectively, is considered to be part of the string or wide-string.

Returns

3 The **strrchr** type-generic macro returns a pointer to the character or wide-character, respectively, or a null pointer if `c` does not occur in the string or wide-string.

7.24.5.6 The `strspn` type-generic macro

Synopsis

1
```
#include <string.h>
```

```
~~size_t strspn(const char *s1, const char *s2);~~
size_t strspn(C *s1, D *s2);
```

Description

2 The **strspn** type-generic macro computes the length of the maximum initial segment of the string or wide-string pointed to by `s1` which consists entirely of characters or wide-characters, respectively, from the string or wide-string pointed to by `s2`.

Returns

3 The **strspn** type-generic macro returns the length of the segment.

7.24.5.7 The strstr type-generic macro

Synopsis

1
```
#include <string.h>
~~char *strstr(const char *s1, const char *s2);~~
C *strstr(C *s1, D *s2)
        [[core::alias(s1)]];
```

Description

2 The **strstr** type-generic macro locates the first occurrence in the string or wide-string pointed to by `s1` of the sequence of characters or wide-characters, respectively, (excluding the terminating null character or wide-character) in the string or wide-string pointed to by `s2`.

Returns

3 The **strstr** type-generic macro returns a pointer to the located string or wide-string, or a null pointer if the string or wide-string is not found. If `s2` points to a string or wide-string with zero length, the function returns `s1`.

7.24.5.8 The strtok type-generic macro

Synopsis

1
```
#include <string.h>
~~char *strtok(char * restrict s1, const char * restrict s2);~~
C *strtok(C * [[core::noalias]] s1, D * [[core::noalias]] s2)
        [[core::alias(s1)]];
```

Constraints

2 The first argument to a call shall be a pointer to character type or **wchar_t** that is not **const** qualified. The second shall be the same type as the first, only that the pointed-to type may be qualified differently.

Description

3 A sequence of calls to the **strtok** type-generic macro breaks the string or wide-string pointed to by `s1` into a sequence of tokens, each of which is delimited by a character or wide-character, respectively, from the string or wide-string pointed to by `s2`. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string or wide-string pointed to by `s2` may be different from call to call.

4 The first call in the sequence searches the string or wide-string pointed to by `s1` for the first character or wide-character, respectively, that is *not* contained in the current separator string or wide-string pointed to by `s2`. If no such character or wide-character is found, then there are no tokens in the string or wide-string pointed to by `s1` and the **strtok** type-generic macro returns a null pointer. If such a character or wide-character is found, it is the start of the first token.

5 The **strtok** type-generic macro then searches from there for a character or wide-character, respectively, that *is* contained in the current separator string or wide-string. If no such character or wide-character is found, the current token extends to the end of the string or wide-string

pointed to by `s1`, and subsequent searches for a token will return a null pointer. If such a character or wide-character is found, it is overwritten by a null character or wide-character, which terminates the current token. The **strtok** type-generic macro saves a pointer to the following character or wide-character, from which the next search for a token will start.

6 Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

7 The **strtok** type-generic macro is not required to avoid data races with other calls to the **strtok** type-generic macro. The implementation shall behave as if no library function calls the **strtok** type-generic macro.

Returns

8 The **strtok** type-generic macro returns a pointer to the first character or wide-character, respectively, of a token, or a null pointer if there is no token.

9 EXAMPLE

```
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");         // t points to the token "a"
t = strtok(NULL, ","); // t points to the token "??b"
t = strtok(NULL, "#,"); // t points to the token "c"
t = strtok(NULL, "?"); // t is a null pointer
t = strtok(nullptr, ",");     // t points to the token "??b"
t = strtok(nullptr, "#,");    // t points to the token "c"
t = strtok(nullptr, "?");     // t is a null pointer
```

~~The **strtok_s** function (??).~~

7.24.6 Miscellaneous functions

7.24.6.1 The tovoidptr type-generic macro

Synopsis

1
```
#include <string.h>
void *memset(void *s, int c, size_t n);
V *tovoidptr(C *s) [[core::alias(s)]];
```

Constraints

2 The argument shall have pointer to object type.

Description

3 The **tovoidptr** type-generic macro converts its argument to a **void** pointer. The return type `V` is the version of **void** that has the same qualifications as `C`.

Returns

4 The **tovoidptr** type-generic macro returns the converted value of `s`.

5 NOTE A possible implementation of the **tovoidptr** type-generic macro is `(true ? s : (void*)1)`.

7.24.6.2 The memset type-generic macro

Synopsis

1
```
#include <string.h>
C *memset(C *[[core::writethrough]] s, int c, size_t n) [[core::alias(s)]];
```

Constraints

2 The first argument shall have pointer to object type.

Description

3 The `memset` type-generic macro copies the value of `c` (converted to an **`unsigned char`**) into each of the first `n` ~~characters of the object~~ bytes of the byte array pointed to by `s`.

4 The `memset` type-generic macro behaves as if the first argument was first implicitly converted to a **`void`** pointer as if by `tovoidptr`.

Returns

5 The `memset` type-generic macro returns the value of `s`.

6 **NOTE** Some library implementations are able to integrate calls to `memset` type-generic macro very closely into their callers, such that they take the effective type of the object that is passed as first argument into account. This can lead to executables that are optimized very effectively. On the other hand such informed handling of byte arrays can lead to leaks of sensible information between different translation units, that cannot be assessed automatically by a translator. Applications that deal with such sensible information should use the variant of this interface that has `C` as **`void volatile`**, such that the writes to the individual bytes of the array are performed unconditionally.

7.24.6.3 The `strerror` function

Synopsis

1
```
#include <string.h>
char *strerror(int errnum);
const char * [[core::alias]] strerror(int errnum) [[core::evaluates(locale)]];
```

Description

2 The `strerror` function maps the number in `errnum` to a message string. Typically, the values for `errnum` come from `errno`, but `strerror` shall map any value of type **`int`** to a message.

3 The `strerror` function is not required to avoid data races with other calls to the `strerror` function.The implementation shall behave as if no library function calls the `strerror` function.

Returns

4 The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function. ~~The `strerror_s` function (??).~~

7.24.6.4 The `strlen` type-generic macro

Synopsis

1
```
#include <string.h>
        size_t strlen(const char *s);
size_t strlen(A x)
      [[ core::stateless, core::idempotent ]] ;
size_t strlen(A x, UT baseflags)
      [[ core::stateless, core::idempotent ]] ;
size_t strlen(A x, UT baseflags, const char* [[ core::noalias ]] glue)
      [[ core::stateless, core::idempotent ]] ;
```

Constraints

2 The arguments shall have same types and fulfill constraints as for the correponding arguments to the **totext** type-generic macro. The same **core** attributes and the **constexpr** specifier are implied as for corresponding calls to the **totext** type-generic macro.

Description

3 ~~The~~ When called with arguments, the **strlen** type-generic macro computes the length of the ~~string pointed to by~~ s textual representation of x as by calls to **totext**, respectively:

```
  totext(0, nullptr, x)
  totext(0, nullptr, x, baseflags)
  totext(0, nullptr, x, baseflags, glue)
```

4 An evaluation of the **strlen** type-generic macro without arguments results in a function pointer of the form

```
size_t (*)(const char* x)
```

When it is called with an appropriate argument, such a function pointer behaves like the macro.

Returns

5 The **strlen** type-generic macro returns the number of characters that ~~precede the terminating null character~~a full textual representation of its first argument by means of a call to **totext** would produce when presented with a buffer that is large enough, or **SIZE_MAX** if an encoding error occurs. Interaction with **errno** and **locale** are as described for **totext**.

6 If evaluated without arguments, the **strlen** type-generic macro returns a function pointer as indicated above.

7.24.6.5 The `strdup` type-generic macro

Synopsis

1
```
#include <string.h>
        char *strdup(const char *s);
char* strdup(A x)
            [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* strdup(A x, UT baseflags)
            [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* strdup(A x, UT baseflags, const char* [[ core::noalias ]] glue)
            [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
```

Constraints

2 The arguments shall have same types and fulfill constraints as for the correponding arguments to the **totext** type-generic macro and the **constexpr** specifier is implied according to the the same rules. If *A* is a pointer type and according to the rules of **totext** the pointed-to object will be accessed, a **core::noalias** attribute for x shall be implied. If the corresponding conversion

implies a wide-character or wide-string to multi-byte conversion, **core::modifies(errno)** and **core::evaluates(locale)** attributes shall be implied.

Description

3 The **strdup** type-generic macro creates a ~~copy of the string pointed to by s in a space~~ string as if by a call to **malloc** that holds the textual representation of x as described for **totext**. ~~allocated as if by~~ It behaves as if first a call to ~~**malloc**~~ **strlen** is issued to compute the length n of the text conversion, **malloc** is called to provide a buffer of size n+1, **totext** is called with that buffer, and the pointer to the buffer is returned.

4 The interactions with **locale** and **errno** are the same as for **totext**, as are the rules for the exposure of a storage instance for which the textual representation of a pointer value is stored.

5 An evaluation of the **strdup** type-generic macro without arguments results in a function pointer of the type

```
char* (*)(const char* x)
```

When it is called with an appropriate argument, such a function pointer behaves like the macro.

Returns

6 ~~The~~ If called with arguments, the **strdup** type-generic macro returns a pointer to the first character of the ~~duplicate~~ created string. The returned pointer can be passed to **free**. ~~If no space~~ If no storage instance can be allocated, or if an encoding error occured the **strdup** type-generic macro returns a null pointer. Additionally, if an encoding error occured **errno** is set to **EILSEQ**.

7 If evaluated without arguments, the **strdup** type-generic macro returns a function pointer as indicated above.

8 **NOTE 1** The **strdup** type-generic macro extends a POSIX function of the same name that has recently been adopted into C. The functionality is maintained for the case that *A* is a character pointer, namely the string that is expected there is copied into the freshly created buffer.

9 **NOTE 2** The **strdup** type-generic macro generally does not have the same **core** attributes as a call to **totext**, in particular it cannot be idempotent or even unsequenced because of its impact to the allocation state of the execution.

7.24.6.6 The strndup type-generic macro

Synopsis

1
```
#include <string.h>
~~char *strndup(const char *s, size_t size);~~
char* [[ core::noalias ]] strndup(A x, size_t len)
                    [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* [[ core::noalias ]] strndup(A x, size_t len, UT baseflags)
                    [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* [[ core::noalias ]] strndup(A x, size_t len, UT baseflags,
          const char* [[ core::noalias ]] glue)
                    [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
```

Constraints

2 The arguments shall have same types and fulfill constraints as for the correponding arguments to the **totext** type-generic macro and the **constexpr** specifier is implied according to the the same rules. If *A* is a pointer type and according to the rules of **totext** the pointed-to object will be accessed, a **core::noalias** attribute for x shall be implied. If the corresponding conversion implies a wide-character or wide-string to multi-byte conversion, **core::modifies(errno)** and **core::evaluates(locale)** attributes shall be implied.

Description

3 The **strndup** type-generic macro ~~creates a string initialized with no more than size initial characters of the array pointed to by s and up to the first null character, whichever comes first, in a space allocated~~ creates a string as if by a call to **malloc**. ~~If the array pointed to by s does not contain a null within the first size characters, a null is appended to the copy of the array~~ **malloc**

that holds the textual representation of `x` as described for **`totext`** where `len` serves as an upper bound for the length of the string. If necessary, `len` is adjusted to be positive.

4 The interactions with **`locale`** and **`errno`** are the same as for **`totext`**, as are the rules for the exposure of a storage instance for which the textual representation of a pointer value is stored.

5 An evaluation of the **`strndup`** type-generic macro without arguments results in a function pointer of the type

```
char* (*)(const char* x, size_t len)
```

When it is called with appropriate arguments, such a function pointer behaves like the macro.

6 In all other aspects, the **`strndup`** type-generic macro behaves analogously to the **`strdup`** type-generic macro.

Returns

7 ~~The~~ If called with arguments, the **`strndup`** type-generic macro returns a pointer to the first character of the created string. The returned pointer can be passed to **`free`**. ~~If no space~~ If no storage instance can be allocated or if an encoding error occured the **`strndup`** type-generic macro returns a null pointer. Additionally, if an encoding error occured **`errno`** is set to **`EILSEQ`**.

8 If evaluated without arguments, the **`strndup`** type-generic macro returns a function pointer as indicated above.

9 **NOTE** The **`strndup`** type-generic macro extends a POSIX function of the same name that has recently been adopted into C. The functionality is maintained for the case that *A* is a character pointer, namely the string that is expected there is copied into the freshly created buffer.

7.25 Type-generic math `<tgmath.h>`

1 The obsolescent header `<tgmath.h>` ~~includes the headers and and defines several type-generic macros.~~

~~The feature test macro `__STDC_VERSION_TGMATH_H__` expands to the token *202005*L.~~

~~Of the and functions without an **f** (**float**) or **l** (**long double**) suffix, several have one or more parameters whose corresponding real type is **double**. For each such function, except **modf**, there is a corresponding .The parameters whose corresponding real type is **double** in the function synopsis are . Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.~~

~~Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:~~

> ~~First, if any argument for generic parameters has type **long double**, the type determined is **long double**. Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double**. Otherwise, the type determined is **float**.~~

~~For each unsuffixed function in for which there is a function in with the same name except for a **c** prefix, the corresponding type-generic macro (for both functions) has the same name as the function in . The corresponding type-generic macro for **fabs** and **cabs** is **fabs**.~~ defines no useful features.

~~lll~~

2 **NOTE** For the C/C++ core, the type-generic
~~function function macro~~
~~**acos cacos acos**~~
~~**asin casin asin**~~
~~**atan catan atan**~~
~~**acosh cacosh acosh**~~
~~**asinh casinh asinh**~~
~~**atanh catanh atanh**~~
~~**cos ccos cos**~~
~~**sin csin sin**~~
~~**tan ctan tan**~~
~~**cosh ccosh cosh**~~
~~**sinh csinh sinh**~~
~~**tanh ctanh tanh**~~
~~**exp cexp exp**~~
~~**log clog log**~~
~~**pow cpow pow**~~
~~**sqrt csqrt sqrt**~~
~~**fabs cabs fabs**~~

~~If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.~~

~~For each unsuffixed function in without a **c** -prefixed counterpart in~~ features that C provides through this header are integrated in the `<math.h>` and `<complex.h>` ~~(except **modf**), the corresponding type-generic macro has the same name as the function. These type-generic macros are: **atan2**~~
~~**cbrt**~~
~~**ceil**~~
~~**copysign**~~
~~**erf**~~
~~**erfc**~~
~~**exp2**~~
~~**expm1**~~
~~**fdim**~~
~~**floor**~~
~~**fma**~~
~~**fmax**~~
~~**fmin**~~
~~**fmod**~~
~~**frexp**~~
~~**hypot**~~
~~**ilogb**~~
~~**ldexp**~~

~~lgamma~~
~~llrint~~
~~llround~~
~~log10~~
~~log1p~~
~~log2~~
~~logb~~
~~lrint~~
~~lround~~
~~nearbyint~~
~~nextafter~~
~~nexttoward~~
~~remainder~~
~~remquo~~
~~rint~~
~~round~~
~~scalbn~~
~~scalbln~~
~~tgamma~~
~~trunc If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.~~

~~For each unsuffixed function in that is not a c -prefixed counterpart to a function in , the corresponding type-generic macro has the same name as the function. These type-generic macros are : carg~~
~~cimag~~
~~conj~~
~~cproj~~
~~creal~~

~~Use of the macro with any real or complex argument invokes a complex function.~~

~~The type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows: where where _Roundwise_cbrtl, _Roundwise_cbrt, and _Roundwise_cbrtf are pointers to functions that are equivalent to cbrtl, cbrt, and cbrtf, respectively, but that are guaranteed to be affected by constant rounding modes (??)~~headers.

~~With the declarations functions invoked by use of type-generic macros are shown in the following table:~~

~~llmacro useinvokes~~
~~exp(n) exp(n), the function~~
~~acosh(f) acoshf(f)~~
~~sin(d) sin(d), the function~~
~~atan(ld) atanl(ld)~~
~~log(fc) clogf(fc)~~
~~sqrt(dc) csqrt(dc)~~
~~pow(ldc, f) cpowl(ldc, f)~~
~~remainder(n, n) remainder(n, n), the function~~
~~nextafter(d, f) nextafter(d, f), the function~~
~~nexttoward(f, ld) nexttowardf(f, ld)~~
~~copysign(n, ld) copysignl(n, ld)~~
~~ceil(fc) undefined behavior~~
~~rint(dc) undefined behavior~~
~~fmax(ldc, ld) undefined behavior~~
~~carg(n) carg(n), the function~~
~~cproj(f) cprojf(f)~~
~~creal(d) creal(d), the function~~
~~cimag(ld) cimagl(ld)~~
~~fabs(fc) cabsf(fc)~~
~~carg(dc) carg(dc), the function~~
~~cproj(ldc) cprojl(ldc)~~

7.26 Threads `<threads.h>`

7.26.1 Introduction

1 The header `<threads.h>` includes the header `<time.h>`, defines ~~macros~~a macro, and declares types, enumeration constants, and functions that support multiple threads of execution.[419)]

2 Implementations that define the macro `__STDC_NO_THREADS__` need not provide this header nor support any of its facilities.

~~which expands to the keyword `_Thread_local`; which expands to a value that can be used to initialize an object of type~~ `once_flag`~~; 3and~~ The macros is

```
TSS_DTOR_ITERATIONS
```

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

4 The types are

```
cnd_t
```

which is a complete opaque object type that holds an identifier for a condition variable;

```
thrd_t
```

which is a complete object type that holds an identifier for a thread;

```
tss_t
```

which is a complete object type that holds an identifier for a thread-specific storage pointer;

```
mtx_t
```

which is a complete opaque object type that holds an identifier for a mutex;

```
tss_dtor_t
```

which is the function pointer type `void (*)(void*)`, used for a destructor for a thread-specific storage pointer;

```
thrd_start_t
```

which is the function pointer type `int (*)(void*)` that is passed to `thrd_create` to create a new thread; and

```
once_flag
```

which is a complete opaque object type that holds a flag for use by `call_once`.

5 The enumeration constants are

```
mtx_plain
```

which is passed to `mtx_init` to create a mutex object that does not support timeout;

```
mtx_recursive
```

which is passed to `mtx_init` to create a mutex object that supports recursive locking;

419)See "future library directions" (7.32.14).

```
mtx_timed
```

which is passed to **mtx_init** to create a mutex object that supports timeout;

```
thrd_timedout
```

which is returned by a timed wait function to indicate that the time specified in the call was reached without acquiring the requested resource;

```
thrd_success
```

which is returned by a function to indicate that the requested operation succeeded;

```
thrd_busy
```

which is returned by a function to indicate that the requested operation failed because a resource requested by a test and return function is already in use;

```
thrd_error
```

which is returned by a function to indicate that the requested operation failed; and

```
thrd_nomem
```

which is returned by a function to indicate that the requested operation failed because it was unable to allocate memory.

Forward references: date and time (7.27).

7.26.2 Initialization functions

7.26.2.1 The `call_once` function

Synopsis

1
```
#include <threads.h>
void call_once(once_flag *flag, void (*func)(void));
```

Description

2 The **call_once** function uses the **once_flag** pointed to by `flag` to ensure that `func` is called exactly once, the first time the **call_once** function is called with that value of `flag`. Completion of an effective call to the **call_once** function synchronizes with all subsequent calls to the **call_once** function with the same value of `flag`. If the **once_flag** is not initialized, the behavior is undefined.[420)]

Returns

3 The **call_once** function returns no value.

7.26.3 Condition variable functions

1 Objects of type **cnd_t** that are used with the functions **cnd_broadcast**, **cnd_destroy**, **cnd_signal**, **cnd_timedwait** and **cnd_wait** shall be properly initialized by explicit or implicit default initialization or by a call to **cnd_init**. Once **cnd_destroy** has called on an object of type **cnd_t** it is in an indeterminate state and a race-free call to **cnd_init** shall be used to re-initialize it before it can be used race-free with any of the other functions. When called properly as indicated in the respective clauses, the functions **cnd_broadcast**, **cnd_signal**, **cnd_timedwait** and **cnd_wait** shall not produce race conditions.

2 **NOTE 1** The current C standard only allows dynamic initialization of **cnd_t** by means of calls to **cnd_init**.

420) As **once_flag** is an opaque object type, the only possibilities to initialize such an object are implicit or explicit default initialization.

7.26.3.1 The cnd_broadcast function

Synopsis

1
```
#include <threads.h>
int cnd_broadcast(cnd_t *cond);
```

Description

2 The **cnd_broadcast** function unblocks all of the threads that are blocked on the condition variable pointed to by cond at the time of the call. If no threads are blocked on the condition variable pointed to by cond at the time of the call, the function does nothing.

Returns

3 The **cnd_broadcast** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.26.3.2 The cnd_destroy function

Synopsis

1
```
#include <threads.h>
void cnd_destroy(cnd_t *cond);
```

Description

2 The **cnd_destroy** function releases all resources used by the condition variable pointed to by cond. The **cnd_destroy** function requires that no threads be blocked waiting for the condition variable pointed to by cond.

Returns

3 The **cnd_destroy** function returns no value.

7.26.3.3 The cnd_init function

Synopsis

1
```
#include <threads.h>
int cnd_init(cnd_t *cond);
int cnd_init(cnd_t * [[ core::writethrough ]] cond);
```

Description

2 The **cnd_init** function ~~creates~~ initializes a condition variable within *cond and is equivalent to an implicit or explicit default initialization of that object. If it succeeds it sets the ~~variable~~ object pointed to by cond to a ~~value~~ state that uniquely identifies the newly created condition variable. A thread that calls **cnd_wait** on a newly created condition variable will block.

Returns

3 The **cnd_init** function returns **thrd_success** on success, or **thrd_nomem** if no memory could be allocated for the newly created condition, or **thrd_error** if the request could not be honored.

7.26.3.4 The cnd_signal function

Synopsis

1
```
#include <threads.h>
int cnd_signal(cnd_t *cond);
```

Description

2 The **cnd_signal** function unblocks one of the threads that are blocked on the condition variable pointed to by cond at the time of the call. If no threads are blocked on the condition variable at the time of the call, the function does nothing and returns success.

Returns

3 The **cnd_signal** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.26.3.5 The `cnd_timedwait` function

Synopsis

1
```
#include <threads.h>
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
     const struct timespec *restrict ts);
```

Description

2 The `cnd_timedwait` function atomically unlocks the mutex pointed to by `mtx` and blocks until the condition variable pointed to by `cond` is signaled by a call to `cnd_signal` or to `cnd_broadcast`, or until after the `TIME_UTC`-based calendar time pointed to by `ts`, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the variable pointed to by `mtx` before it returns. The `cnd_timedwait` function requires that the mutex pointed to by `mtx` be locked by the calling thread.

Returns

3 The `cnd_timedwait` function returns `thrd_success` upon success, or `thrd_timedout` if the time specified in the call was reached without acquiring the requested resource, or `thrd_error` if the request could not be honored.

7.26.3.6 The `cnd_wait` function

Synopsis

1
```
#include <threads.h>
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Description

2 The `cnd_wait` function atomically unlocks the mutex pointed to by `mtx` and blocks until the condition variable pointed to by `cond` is signaled by a call to `cnd_signal` or to `cnd_broadcast`, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the mutex pointed to by `mtx` before it returns. The `cnd_wait` function requires that the mutex pointed to by `mtx` be locked by the calling thread.

Returns

3 The `cnd_wait` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

7.26.4 Mutex functions

1 ~~For purposes of determining the existence of a data race, lock and unlock operations behave as atomic operations. All lock~~Objects of type `mtx_t` that are used with the functions `mtx_destroy`, `mtx_lock`, `mtx_timedlock`, `mtx_trylock`, and `mtx_unlock` shall be properly initialized by explicit or implicit default initialization or by a call to `mtx_init`. Once `mtx_destroy` has been called on an object of type `mtx_t` it is in an indeterminate state and a race-free call to `mtx_init` shall be used to re-initialize it before it can be used race-free with any of the other functions. When called properly as indicated in the respective clauses, the functions `cnd_timedwait`, `cnd_wait`, `mtx_lock`, `mtx_timedlock`, `mtx_trylock`, and `mtx_unlock` shall not produce race conditions with respect to their `mtx_t` object. To determine the happened-before relation, lock, wait and unlock operations on ~~a particular mutex occur in some~~ the same mutex object synchronize and they appear in a particular total order ~~.~~ for that mutex.

2 **NOTE 1** This total order can be viewed as the modification order of the mutex.

3 **NOTE 2** The current C standard only allows dynamic initialization of `mtx_t` by means of calls to `mtx_init`.

7.26.4.1 The `mtx_destroy` function

Synopsis

1
```
#include <threads.h>
void mtx_destroy(mtx_t *mtx);
```

Description

2 The **mtx_destroy** function releases any resources used by the mutex pointed to by `mtx`. No threads can be blocked waiting for the mutex pointed to by `mtx`.

Returns

3 The **mtx_destroy** function returns no value.

7.26.4.2 The mtx_init function

Synopsis

1
```
#include <threads.h>
int mtx_init(mtx_t *mtx, int type);
int mtx_init(mtx_t * [[core::writethrough]] mtx, int type);
```

Description

2 The **mtx_init** function creates a mutex object with properties indicated by `type`, which shall have one of these values:

mtx_plain for a simple non-recursive mutex, which is the same as if `*mtx` had been explicitly or implicitly default initialized,

mtx_timed for a non-recursive mutex that supports timeout,

mtx_plain | mtx_recursive for a simple recursive mutex, or

mtx_timed | mtx_recursive for a recursive mutex that supports timeout.

3 If the **mtx_init** function succeeds, it sets the mutex pointed to by `mtx` to a value that uniquely identifies the newly created mutex.

Returns

4 The **mtx_init** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.26.4.3 The mtx_lock function

Synopsis

1
```
#include <threads.h>
int mtx_lock(mtx_t *mtx);
```

Description

2 The **mtx_lock** function blocks until it locks the mutex pointed to by `mtx`. If the mutex is non-recursive, it shall not be locked by the calling thread. Prior calls to **mtx_unlock** on the same mutex synchronize with this operation.

Returns

3 The **mtx_lock** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.26.4.4 The mtx_timedlock function

Synopsis

1
```
#include <threads.h>
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
```

Description

2 The **mtx_timedlock** function endeavors to block until it locks the mutex pointed to by `mtx` or until after the **TIME_UTC**-based calendar time pointed to by `ts`. The specified mutex shall support

timeout. If the operation succeeds, prior calls to **mtx_unlock** on the same mutex synchronize with this operation.

Returns

3 The **mtx_timedlock** function returns **thrd_success** on success, or **thrd_timedout** if the time specified was reached without acquiring the requested resource, or **thrd_error** if the request could not be honored.

7.26.4.5 The mtx_trylock function

Synopsis

1
```
#include <threads.h>
int mtx_trylock(mtx_t *mtx);
```

Description

2 The **mtx_trylock** function endeavors to lock the mutex pointed to by mtx. If the mutex is already locked, the function returns without blocking. If the operation succeeds, prior calls to **mtx_unlock** on the same mutex synchronize with this operation.

Returns

3 The **mtx_trylock** function returns **thrd_success** on success, or **thrd_busy** if the resource requested is already in use, or **thrd_error** if the request could not be honored. **mtx_trylock** may spuriously fail to lock an unused resource, in which case it returns **thrd_busy**.

7.26.4.6 The mtx_unlock function

Synopsis

1
```
#include <threads.h>
int mtx_unlock(mtx_t *mtx);
```

Description

2 The **mtx_unlock** function unlocks the mutex pointed to by mtx. The mutex pointed to by mtx shall be locked by the calling thread.

Returns

3 The **mtx_unlock** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.26.5 Thread functions

7.26.5.1 The thrd_create function

Synopsis

1
```
#include <threads.h>
~~int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);~~
int thrd_create(thrd_t * [[ core::noalias, core::writethrough ]] thr, thrd_start_t func, void *arg);
```

Description

2 The **thrd_create** function creates a new thread executing func(arg). If the **thrd_create** function succeeds, it sets the object pointed to by thr to the identifier of the newly created thread. (A thread's identifier may be reused for a different thread once the original thread has exited and either been detached or joined to another thread.) The completion of the **thrd_create** function synchronizes with the beginning of the execution of the new thread.

3 Returning from func has the same behavior as invoking **thrd_exit** with the value returned from func.

Returns

4 The **thrd_create** function returns **thrd_success** on success, or **thrd_nomem** if no memory could be allocated for the thread requested, or **thrd_error** if the request could not be honored.

7.26.5.2 The thrd_current function

Synopsis

1
```
#include <threads.h>
thrd_t thrd_current(void);
```

Description

2 The **thrd_current** function identifies the thread that called it.

Returns

3 The **thrd_current** function returns the identifier of the thread that called it.

7.26.5.3 The thrd_detach function

Synopsis

1
```
#include <threads.h>
int thrd_detach(thrd_t thr);
```

Description

2 The **thrd_detach** function tells the operating system to dispose of any resources allocated to the thread identified by thr when that thread terminates. The thread identified by thr shall not have been previously detached or joined with another thread.

Returns

3 The **thrd_detach** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.26.5.4 The thrd_equal function

Synopsis

1
```
#include <threads.h>
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

2 The **thrd_equal** function will determine whether the thread identified by thr0 refers to the thread identified by thr1.

Returns

3 The **thrd_equal** function returns zero if the thread thr0 and the thread thr1 refer to different threads. Otherwise the **thrd_equal** function returns a nonzero value.

7.26.5.5 The thrd_exit function

Synopsis

1
```
#include <threads.h>
_Noreturn void thrd_exit(int res);
```

Description

2 For every thread-specific storage key which was created with a non-null destructor and for which the value is non-null, **thrd_exit** sets the value associated with the key to a null pointer value and then ~~invokes~~ calls the destructor with its previous value. ~~The order in which destructors are invoked is unspecified.~~ These destructor calls are indeterminately sequenced.

3 If after this process there remain keys with both non-null destructors and values, the implementation repeats this process up to **TSS_DTOR_ITERATIONS** times.

4 Following this, the **thrd_exit** function terminates execution of the calling thread T and sets its result code to `res`. Finally, there is a sequence point that synchronizes with the completion of a successful call, if any, of the **thrd_join** function for T and with the first call to **atexit** or **at_quick_exit** handlers at program termination, if any.[421)]

5 The program terminates normally after the last thread has been terminated. The behavior is as if the program called the **exit** function with the status **EXIT_SUCCESS** at thread termination time.

Returns

6 The **thrd_exit** function returns no value.

7.26.5.6 The thrd_join function

Synopsis

1
```
#include <threads.h>
int thrd_join(thrd_t thr, int *res);
```

Description

2 The **thrd_join** function joins the thread identified by `thr` with the current thread by blocking until the other thread has terminated. If the parameter `res` is not a null pointer, it stores the thread's result code in the integer pointed to by `res`. The termination of the other thread synchronizes with the completion of the **thrd_join** function. The thread identified by `thr` shall not have been previously detached or joined with another thread.

Returns

3 The **thrd_join** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.26.5.7 The thrd_sleep function

Synopsis

1
```
#include <threads.h>
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
```

Description

2 The **thrd_sleep** function suspends execution of the calling thread until either the interval specified by `duration` has elapsed or a signal which is not being ignored is received. If interrupted by a signal and the `remaining` argument is not null, the amount of time remaining (the requested interval minus the time actually slept) is stored in the interval it points to. The `duration` and `remaining` arguments may point to the same object.

3 The suspension time may be longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than that specified, as measured by the system clock **TIME_UTC**.

Returns

4 The **thrd_sleep** function returns zero if the requested time has elapsed, -1 if it has been interrupted by a signal, or a negative value (which may also be -1) if it fails.

7.26.5.8 The thrd_yield function

Synopsis

1

421) This leaves it unspecified if threads that are terminated by other means than **thrd_exit**, for example by an implementation specific mechanism or because they have not been terminated explicitly before program termination, synchronize with **atexit** or **at_quick_exit** handlers.

```
#include <threads.h>
void thrd_yield(void);
```

Description

2 The **thrd_yield** function endeavors to permit other threads to run, even if the current thread would ordinarily continue to run.

Returns

3 The **thrd_yield** function returns no value.

7.26.6 Thread-specific storage functions

7.26.6.1 The tss_create function

Synopsis

1
```
#include <threads.h>
int tss_create(tss_t *key, tss_dtor_t dtor);
int tss_create(tss_t * [[core::writethrough]] key, tss_dtor_t dtor);
```

Description

2 The **tss_create** function creates a thread-specific storage pointer with destructor `dtor`, which may be null.

3 A null pointer value is associated with the newly created key in all existing threads. Upon subsequent thread creation, the value associated with all keys is initialized to a null pointer value in the new thread.

4 Destructors associated with thread-specific storage are not invoked at program termination.

5 The **tss_create** function shall not be called from within a destructor.

Returns

6 If the **tss_create** function is successful, it sets the thread-specific storage pointed to by `key` to a value that uniquely identifies the newly created pointer and returns **thrd_success**; otherwise, **thrd_error** is returned and the thread-specific storage pointed to by `key` is set to an indeterminate value.

7.26.6.2 The tss_delete function

Synopsis

1
```
#include <threads.h>
void tss_delete(tss_t key);
```

Description

2 The **tss_delete** function releases any resources used by the thread-specific storage identified by `key`. The **tss_delete** function shall only be called with a value for `key` that was returned by a call to **tss_create** before the thread commenced executing destructors.

3 If **tss_delete** is called while another thread is executing destructors, whether this will affect the number of invocations of the destructor associated with `key` on that thread is unspecified.

4 Calling **tss_delete** will not result in the invocation of any destructors.

Returns

5 The **tss_delete** function returns no value.

7.26.6.3 The tss_get function

Synopsis

1
```
#include <threads.h>
void *tss_get(tss_t key);
```

Description

2 The **tss_get** function returns the value for the current thread held in the thread-specific storage identified by `key`. The **tss_get** function shall only be called with a value for `key` that was returned by a call to **tss_create** before the thread commenced executing destructors.

Returns

3 The **tss_get** function returns the value for the current thread if successful, or zero if unsuccessful.

7.26.6.4 The tss_set function

Synopsis

1
```
#include <threads.h>
int tss_set(tss_t key, void *val);
```

Description

2 The **tss_set** function sets the value for the current thread held in the thread-specific storage identified by `key` to `val`. The **tss_set** function shall only be called with a value for `key` that was returned by a call to **tss_create** before the thread commenced executing destructors.

3 This action will not invoke the destructor associated with the key on the value being replaced.

4 If `val` is a valid pointer, its provenance is is henceforth exposed.

Returns

5 The **tss_set** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.27 Date and time <time.h>

7.27.1 Components of time

1 The header `<time.h>` defines several macros, and declares types and functions for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

2 The feature test macro `__CORE_VERSION_TIME_H__` expands to the token `202005L`. The other macros defined are

```
CLOCKS_PER_SEC
```

which expands to an expression with type `clock_t` (described below) that is the number per second of the value returned by the `clock` function; and

```
TIME_UTC
```

which expands to an integer constant greater than 0 that designates the UTC time base.[422)]

3 The types declared are

```
clock_t
```

and

```
time_t
```

which are real types capable of representing times;

```
struct timespec
```

which holds an interval specified in seconds and nanoseconds (which may represent a calendar time based on a particular epoch); and

```
struct tm
```

which holds the components of a calendar time, called the *broken-down time*.

4 The range and precision of times representable in `clock_t` and `time_t` are implementation-defined. The `timespec` structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[423)]

```
time_t tv_sec;  // whole seconds -- ≥ 0
long   tv_nsec; // nanoseconds -- [0, 999999999]
```

The `tm` structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[424)]

```
int tm_sec;   // seconds after the minute -- [0, 60]
int tm_min;   // minutes after the hour -- [0, 59]
int tm_hour;  // hours since midnight -- [0, 23]
int tm_mday;  // day of the month -- [1, 31]
int tm_mon;   // months since January -- [0, 11]
```

422) Implementations can define additional time bases, but are only required to support a real time clock based on UTC.

423) The `tv_sec` member is a linear count of seconds and might not have the normal semantics of a `time_t`.

424) The range [0, 60] for `tm_sec` allows for a positive leap second.

```
        int tm_year;  // years since 1900
        int tm_wday;  // days since Sunday -- [0, 6]
        int tm_yday;  // days since January 1 -- [0, 365]
        int tm_isdst; // Daylight Saving Time flag
```

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

7.27.2 Time manipulation functions

1 Most time manipulation functions access a hidden state **time** which represents the platform's specific software and hardware to deal with time and that is not further specified.

Recommended practice

2 It is recommended that all implementation specific function declarations that deal with notions of time similar to this clause are annotated with the appropriate core function attributes. In particular functions that modify time settings (*e.g* adjusting calendar time or setting the timezone) as perceived by the execution, should announce a **core::modifies(time)** attribute.

7.27.2.1 The clock function

Synopsis

1
```
        #include <time.h>
        clock_t clock(void);
        clock_t clock(void) [[core::evaluates(time)]];
```

Description

2 The **clock** function determines the processor time used.

Returns

3 The **clock** function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock** function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available, the function returns the value **(clock_t)**(-1). If the value cannot be represented, the function returns an unspecified value.[425)]

7.27.2.2 The difftime function

Synopsis

1
```
        #include <time.h>
        double difftime(time_t time1, time_t time0);
```

Description

2 The **difftime** function computes the difference between two calendar times: `time1 - time0`.

Returns

3 The **difftime** function returns the difference expressed in seconds as a **double**.

7.27.2.3 The mktime function

Synopsis

1
```
        #include <time.h>
        time_t mktime(struct tm *timeptr);
        time_t mktime([[core::noalias]] struct tm timeptr[1]) [[core::evaluates(time)]];
```

425)This could be due to overflow of the **clock_t** type.

Description

2 The **mktime** function converts the broken-down time, expressed as local time, in the structure pointed to by timeptr into a calendar time value with the same encoding as that of the values returned by the **time** function. The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above.[426] On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

Returns

3 The **mktime** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value (**time_t**)(-1).

4 **EXAMPLE** What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* ... */

time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 1;
time_str.tm_isdst  = -1;
if (mktime(&time_str) == (time_t)(-1))
if (mktime(&time_str) ≡ (time_t)(-1))
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

7.27.2.4 The time function

Synopsis

1

```
#include <time.h>
time_t time(time_t *timer);
time_t time(time_t *timer) [[core::evaluates(time)]];
```

Description

2 The **time** function determines the current calendar time. The encoding of the value is unspecified.

Returns

3 The **time** function returns the implementation's best approximation to the current calendar time. The value (**time_t**)(-1) is returned if the calendar time is not available. If timer is not a null pointer, the return value is also assigned to the object it points to.

7.27.2.5 The timespec_get function

Synopsis

1

```
#include <time.h>
```

[426] Thus, a positive or zero value for **tm_isdst** causes the **mktime** function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value causes it to attempt to determine whether Daylight Saving Time is in effect for the specified time.

```
~~int timespec_get(struct timespec *ts, int base);~~
int timespec_get( [[ core::noalias, core::writethrough ]] struct timespec ts[1], int base)
        [[ core::evaluates(time) ]] ;
```

Description

2 The **timespec_get** function sets the interval pointed to by `ts` to hold the current calendar time based on the specified time base.

3 If `base` is **TIME_UTC**, the **tv_sec** member is set to the number of seconds since an implementation defined *epoch*, truncated to a whole value and the **tv_nsec** member is set to the integral number of nanoseconds, rounded to the resolution of the system clock.[427)]

Returns

4 If the **timespec_get** function is successful it returns the nonzero value `base`; otherwise, it returns zero.

7.27.3 Time conversion functions

1 Functions with a **_r** suffix place the result of the conversion into the buffer referred by `buf` and return that pointer. These functions and the function **strftime** shall not be subject to data races, unless the time or calendar state is changed in a multi-thread execution.[428)]

2 Functions **asctime**, **ctime**, **gmtime**, and **localtime** are the same as their counterparts suffixed with **_r**. In place of the parameter `buf`, these functions use a pointer to a static object and return it: one or two broken-down time structures (for **gmtime** and **localtime**) or an array of **char** (commonly used by **asctime** and **ctime**). Execution of any of the functions that return a pointer to one of these static objects may overwrite the information returned from any previous call to one of these functions that uses the same object. These functions are not reentrant and are not required to avoid data races with each other. The implementation shall behave as if no other library functions call these functions.

7.27.3.1 The asctime functions

Synopsis

1
```
#include <time.h>
~~char *asctime(const struct tm *timeptr);~~
~~char *asctime_r(const struct tm *timeptr, char *buf);~~
char * [[ core::alias ]] asctime( [[ core::noalias ]] const struct tm timeptr[1]);
char *asctime_r( [[ core::noalias ]] const struct tm timeptr[1],
                 [[ core::noalias ]] char buf[26])
           [[ core::alias(buf) ]] ;
```

Description

2 The **asctime** functions convert the broken-down time in the structure pointed to by `timeptr` into a string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
~~char *asctime_r(const struct tm *timeptr, char *buf)~~
char *asctime_r( [[ core::noalias ]] const struct tm timeptr[1],
                 [[ core::noalias, core::writethrough ]] char buf[26]) [[ core::alias(buf) ]]
{
      static const char wday_name[7][3] = {
```

[427)] Although a **struct timespec** object describes times with nanosecond resolution, the available resolution is system dependent and could even be greater than 1 second.

[428)] This does not mean that these functions may not read global state that describes the time and calendar settings of the execution, such as the **LC_TIME** locale or the implementation defined specification of the local time zone. Only the setting of that state by **setlocale** or by means of implementation-defined functions may constitute races.

```
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };

    snprintf(buf, 26, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
        wday_name[timeptr→tm_wday],
        mon_name[timeptr→tm_mon],
        timeptr→tm_mday, timeptr→tm_hour,
        timeptr→tm_min, timeptr→tm_sec,
        1900 + timeptr→tm_year);
    return buf;
}
```

3 If any of the members of the broken-down time contain values that are outside their normal ranges,[429] the behavior of the **asctime** functions is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined. The buf parameter for **asctime_r** shall point to a buffer of at least 26 bytes.

Returns

4 The **asctime** functions return a pointer to the string.

7.27.3.2 The ctime functions

Synopsis

1
```
#include <time.h>
    char *ctime(const time_t *timer);
    char *ctime_r(const time_t *timer, char *buf);
char *ctime( [[core::noalias]] const time_t timer[1])
    [[core::alias(asctime), core::evaluates(locale, time)]];
char *ctime_r( [[core::noalias]] const time_t timer[1], [[core::noalias, core::writethrough]] char buf[26])
    [[core::alias(buf), core::evaluates(locale, time)]];
```

Description

2 The **ctime** functions convert the calendar time pointed to by timer to local time in the form of a string. They are equivalent to

```
    asctime(localtime_r(timer, (struct tm[1]){ 0 }))
```

and

```
    asctime_r(localtime_r(timer, (struct tm[1]){ 0 }), buf)
```

The buf parameter for **ctime_r** shall point to a buffer of at least 26 bytes.

Returns

3 The **ctime** functions return the pointer returned by the **asctime** functions with that broken-down time as argument.

Forward references: the **localtime** functions (7.27.3.4).

[429] See 7.27.1.

7.27.3.3 The gmtime functions

Synopsis

1
```
#include <time.h>
~~struct tm *gmtime(const time_t *timer);~~
~~struct tm *gmtime_r(const time_t *timer, struct tm *buf);~~
struct tm *gmtime( [[core::noalias]] const time_t timer[1])
          [[core::alias(localtime), core::evaluates(time)]];
struct tm *gmtime_r( [[core::noalias]] const time_t timer[1],
                     [[core::noalias, core::writethrough]] struct tm buf[1])
          [[core::alias(buf), core::evaluates(time)]];
```

Description

2 The gmtime functions convert the calendar time pointed to by timer into a broken-down time, expressed as UTC.

Returns

3 The gmtime functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

7.27.3.4 The localtime functions

Synopsis

1
```
#include <time.h>
~~struct tm *localtime(const time_t *timer);~~
~~struct tm *localtime_r(const time_t *timer, struct tm *buf);~~
struct tm * [[core::alias]] localtime( [[core::noalias]] const time_t timer[1])
          [[core::evaluates(locale, time)]];
struct tm *localtime_r( [[core::noalias]] const time_t timer[1],
                     [[core::noalias, core::writethrough]] struct tm buf[1])
          [[core::alias(buf), core::evaluates(locale, time)]];
```

Description

2 The localtime functions converts the calendar time pointed to by timer into a broken-down time, expressed as local time.

Returns

3 The localtime functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

7.27.3.5 The strftime function

Synopsis

1
```
#include <time.h>
~~size_t strftime(char * restrict s, size_t maxsize, const char * restrict format,~~
~~        const struct tm * restrict timeptr);~~
size_t strftime(char * [[core::noalias, core::writethrough]] s, size_t maxsize,
                const char * [[core::noalias]] format,
                const struct tm * [[core::noalias]] timeptr)
          [[core::evaluates(locale)]];
```

Description

2 The strftime function places characters into the array pointed to by s as controlled by the string pointed to by format. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character, possibly followed by an E or 0 modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are

copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array.

3 Each conversion specifier shall be replaced by appropriate characters as described in the following list. The appropriate characters shall be determined using the **`LC_TIME`** category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

- `%a` is replaced by the locale's abbreviated weekday name. [**`tm_wday`**]
- `%A` is replaced by the locale's full weekday name. [**`tm_wday`**]
- `%b` is replaced by the locale's abbreviated month name. [**`tm_mon`**]
- `%B` is replaced by the locale's full month name. [**`tm_mon`**]
- `%c` is replaced by the locale's appropriate date and time representation. [all specified in 7.27.1]
- `%C` is replaced by the year divided by 100 and truncated to an integer, as a decimal number (`00`–`99`). [**`tm_year`**]
- `%d` is replaced by the day of the month as a decimal number (`01`–`31`). [**`tm_mday`**]
- `%D` is equivalent to "`%m/%d/%y`". [**`tm_mon`**, **`tm_mday`**, **`tm_year`**]
- `%e` is replaced by the day of the month as a decimal number (`1`–`31`); a single digit is preceded by a space. [**`tm_mday`**]
- `%F` is equivalent to "`%Y-%m-%d`" (the ISO 8601 date format). [**`tm_year`**, **`tm_mon`**, **`tm_mday`**]
- `%g` is replaced by the last 2 digits of the week-based year (see below) as a decimal number (`00`–`99`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]
- `%G` is replaced by the week-based year (see below) as a decimal number (e.g., 1997). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]
- `%h` is equivalent to "`%b`". [**`tm_mon`**]
- `%H` is replaced by the hour (24-hour clock) as a decimal number (`00`–`23`). [**`tm_hour`**]
- `%I` is replaced by the hour (12-hour clock) as a decimal number (`01`–`12`). [**`tm_hour`**]
- `%j` is replaced by the day of the year as a decimal number (`001`–`366`). [**`tm_yday`**]
- `%m` is replaced by the month as a decimal number (`01`–`12`). [**`tm_mon`**]
- `%M` is replaced by the minute as a decimal number (`00`–`59`). [**`tm_min`**]
- `%n` is replaced by a new-line character.
- `%p` is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock. [**`tm_hour`**]
- `%r` is replaced by the locale's 12-hour clock time. [**`tm_hour`**, **`tm_min`**, **`tm_sec`**]
- `%R` is equivalent to "`%H:%M`". [**`tm_hour`**, **`tm_min`**]
- `%S` is replaced by the second as a decimal number (`00`–`60`). [**`tm_sec`**]
- `%t` is replaced by a horizontal-tab character.
- `%T` is equivalent to "`%H:%M:%S`" (the ISO 8601 time format). [**`tm_hour`**, **`tm_min`**, **`tm_sec`**]
- `%u` is replaced by the ISO 8601 weekday as a decimal number (`1`–`7`), where Monday is 1. [**`tm_wday`**]
- `%U` is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (`00`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]
- `%V` is replaced by the ISO 8601 week number (see below) as a decimal number (`01`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]
- `%w` is replaced by the weekday as a decimal number (`0`–`6`), where Sunday is 0. [**`tm_wday`**]
- `%W` is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (`00`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]

`%x` is replaced by the locale's appropriate date representation. [all specified in 7.27.1]

`%X` is replaced by the locale's appropriate time representation. [all specified in 7.27.1]

`%y` is replaced by the last 2 digits of the year as a decimal number (`00`–`99`). [**`tm_year`**]

`%Y` is replaced by the year as a decimal number (e.g., `1997`). [**`tm_year`**]

`%z` is replaced by the offset from UTC in the ISO 8601 format "`-0430`" (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [**`tm_isdst`**]

`%Z` is replaced by the locale's time zone name or abbreviation, or by no characters if no time zone is determinable. [**`tm_isdst`**]

`%%` is replaced by `%`.

4 Some conversion specifiers can be modified by the inclusion of an `E` or `O` modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

`%Ec` is replaced by the locale's alternative date and time representation.

`%EC` is replaced by the name of the base year (period) in the locale's alternative representation.

`%Ex` is replaced by the locale's alternative date representation.

`%EX` is replaced by the locale's alternative time representation.

`%Ey` is replaced by the offset from `%EC` (year only) in the locale's alternative representation.

`%EY` is replaced by the locale's full alternative year representation.

`%Ob` is replaced by the locale's abbreviated alternative month name.

`%OB` is replaced by the locale's alternative appropriate full month name.

`%Od` is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

`%Oe` is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading spaces).

`%OH` is replaced by the hour (24-hour clock), using the locale's alternative numeric symbols.

`%OI` is replaced by the hour (12-hour clock), using the locale's alternative numeric symbols.

`%Om` is replaced by the month, using the locale's alternative numeric symbols.

`%OM` is replaced by the minutes, using the locale's alternative numeric symbols.

`%OS` is replaced by the seconds, using the locale's alternative numeric symbols.

`%Ou` is replaced by the ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1.

`%OU` is replaced by the week number, using the locale's alternative numeric symbols.

`%OV` is replaced by the ISO 8601 week number, using the locale's alternative numeric symbols.

`%Ow` is replaced by the weekday as a number, using the locale's alternative numeric symbols.

`%OW` is replaced by the week number of the year, using the locale's alternative numeric symbols.

`%Oy` is replaced by the last 2 digits of the year, using the locale's alternative numeric symbols.

5 `%g`, `%G`, and `%V` give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, `%G` is replaced by `1998` and `%V` is replaced by `53`. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, `%G` is replaced by `1998` and `%V` is replaced by `01`.

6 If a conversion specifier is not one of the above, the behavior is undefined.

7 In the `"C"` locale, the `E` and `O` modifiers are ignored and the replacement strings for the following specifiers are:

`%a`	the first three characters of `%A`.
`%A`	one of "`Sunday`", "`Monday`", ..., "`Saturday`".
`%b`	the first three characters of `%B`.
`%B`	one of "`January`", "`February`", ..., "`December`".
`%c`	equivalent to "`%a %b %e %T %Y`".
`%p`	one of "`AM`" or "`PM`".
`%r`	equivalent to "`%I:%M:%S %p`".
`%x`	equivalent to "`%m/%d/%y`".
`%X`	equivalent to `%T`.
`%Z`	implementation-defined.

Returns

8 If the total number of resulting characters including the terminating null character is not more than `maxsize`, the `strftime` function returns the number of characters placed into the array pointed to by `s` not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

7.28 Unicode utilities <uchar.h>

1 The header `<uchar.h>` declares ~~types~~ a macro, a type and functions for manipulating Unicode characters.

2 The ~~types declared are~~ type declared is **`mbstate_t`** (described in 7.29.1)~~and **`size_t`** (described in 7.19); which is an unsigned integer type used for 16-bit characters and is the same type as (described in 7.20.1.1); and which is an unsigned integer type used for 32-bit characters and is the same type as (also described in 7.20.1.1).~~ .

3 The feature test macro **`__CORE_VERSION_UCHAR_H__`** expands to the token *`202005`*`L`.

4 **NOTE** The **`mbrtoc8`** and **`c8rtomb`** functions are extensions by the C++ standard that are not yet part of the C standard. The current C standard allows the types **`char16_t`** and **`char32_t`** to represent other encodings than UTF–16 and UTF–32, respectively. This specification constrains implementations to use the UTF encodings for these types, so the descriptions for the functions in this clause have been aleviated to these particular cases.

7.28.1 Restartable multibyte/wide character conversion functions

1 These functions have a parameter, `ps`, of type pointer to **`mbstate_t`** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence, which the functions alter as necessary. If `ps` is a null pointer, each function uses its own internal **`mbstate_t`** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

7.28.1.1 The mbrtoc8 and mbrtoc16 functions

Synopsis

1
```
#include <uchar.h>
~~size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n,
          mbstate_t * restrict ps);~~
size_t mbrtoc8(char8_t *[[core::noalias, core::writethrough]] pc8,
          const char *[[core::noalias]] s, size_t n, mbstate_t *[[core::noalias]] ps)
                    [[core::modifies(errno)]];
size_t mbrtoc16(char16_t *[[core::noalias, core::writethrough]] pc16,
          const char *[[core::noalias]] s, size_t n, mbstate_t *[[core::noalias]] ps)
                    [[core::modifies(errno)]];
```

Description

2 If `s` is a null pointer, the **`mbrtoc8`** and **`mbrtoc16`** functions ~~is~~ are equivalent to the ~~call~~calls, respectively:

```
~~mbrtoc16(NULL, "", 1, ps)~~
mbrtoc8(nullptr, "", 1, ps)
mbrtoc16(nullptr, "", 1, ps)
```

In this case, the values of the parameters `pc8` or `pc16` and `n` are ignored.

3 If `s` is not a null pointer, the **`mbrtoc8`** and **`mbrtoc16`** functions ~~inspects~~ inspect at most `n` bytes (including any shift sequences) beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character~~(including any shift sequences)~~. If the function determines that the next multibyte character is complete and valid, it determines the ~~values of the corresponding wide characters~~ corresponding universal character name and then, if ~~`pc16`~~ `pc8` or `16` is not a null pointer, ~~stores~~ store the value of the first (or only) such character of the UTF–8 or UTF–16 encoding, respectively, in the object pointed to by `pc8` or `pc16`. Subsequent calls will store successive ~~wide characters~~ characters of the encoding without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **mbrtoc8** and **mbrtoc16** functions ~~returns~~ return the first of the following that applies (given the current conversion state):

0 if the next n or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* n *inclusive* if the next n or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

(size_t)(-3) if the next encoding character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(size_t)(-2) if the next n bytes contribute to an incomplete (but potentially valid) multibyte character, and all n bytes have been processed (no value is stored).[430]

(size_t)(-1) if an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **EILSEQ** is stored in **errno**, and the conversion state is unspecified.

7.28.1.2 The c8rtomb and c16rtomb functions

Synopsis

1
```
#include <uchar.h>
        size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
size_t c8rtomb(char *[[core::noalias, core::writethrough]] s, char8_t c8,
        mbstate_t *[[core::noalias]] ps)
                [[core::modifies(errno)]]
size_t c16rtomb(char *[[core::noalias, core::writethrough]] s, char16_t c16,
        mbstate_t *[[core::noalias]] ps)
                [[core::modifies(errno)]]
```

Description

2 If s is a null pointer, the **c8rtomb** and **c16rtomb** functions ~~is~~ are equivalent to the ~~call~~ calls, respectively,

```
                c8rtomb(buf, L'\0', ps)
                c16rtomb(buf, L'\0', ps)
```

where buf is an internal buffer.

3 If s is not a null pointer, the **c8rtomb** and **c16rtomb** functions ~~determines~~ determine the number of bytes (including any shift sequences) needed to represent the multibyte character that corresponds to the ~~wide character~~ universal character name given or completed by ~~c16 (including any shift sequences), and stores~~ c8 or c16, respectively, and store the multibyte character representation in the array whose first element is pointed to by ~~s, or stores nothing if c16 does~~ s, or store nothing if c8 or c16 do not represent a complete character. At most **MB_CUR_MAX** bytes are stored. ~~If c16 is a null wide~~ If c8 or c16 are a null character, a null ~~byte~~ character is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **c8rtomb** and **c16rtomb** functions ~~returns~~ return the number of bytes ~~stored in the array object~~ (including any shift sequences) ~~. When c16 is~~ stored in the array object. An encoding error

[430] When n has at least the value of the **MB_CUR_MAX** macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

occurs when c8 is greater than 0xFF or c16 is greater than 0xFFFF,[431] when the state is the initial conversion state and c8 is greater than 0x7F or c16 is a low surrogate, or when the state is not the initial conversion state and c8 or c16 are not a valid ~~wide character~~ continuation of a UTF–8 or UTF–16 character sequence, respectively, that lead to that state. In case of such an error, ~~an encoding erroroccurs:~~ the function stores the value of the macro **EILSEQ** in **errno** and returns (**size_t**)(−1); the conversion state is unspecified.

7.28.1.3 The `mbrtoc32` function

Synopsis

1
```
#include <uchar.h>
      size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n,
            mbstate_t * restrict ps);
size_t mbrtoc32(char32_t *[[core::noalias, core::writethrough]] pc32,
      const char *[[core::noalias]] s, size_t n, mbstate_t *[[core::noalias]] ps)
            [[core::modifies(errno)]]
```

Description

2 If `s` is a null pointer, the **mbrtoc32** function is equivalent to the call:

```
            mbrtoc32(NULL, "", 1, ps)
            mbrtoc32(nullptr, "", 1, ps)
```

In this case, the values of the parameters `pc32` and `n` are ignored.

3 If `s` is not a null pointer, the **mbrtoc32** function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes (including any shift sequences) needed to complete the next multibyte character~~(including any shift sequences)~~. If the function determines that the next multibyte character is complete and valid, it determines the ~~values of the corresponding wide characters~~ universal character name and then, if `pc32` is not a null pointer, stores the value ~~of the first (or only) such character~~ in the object pointed to by `pc32`. ~~Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored.~~ If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **mbrtoc32** function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

~~(**size_t**)(−3)~~ ~~if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).~~

(**size_t**)(−2) if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[432]

(**size_t**)(−1) if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **EILSEQ** is stored in **errno**, and the conversion state is unspecified.

[431] This may happen if the **char8_t** or **char16_t** types are wider than 8 or 16, respectively.

[432] When `n` has at least the value of the **MB_CUR_MAX** macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.28.1.4 The `c32rtomb` function

Synopsis

1
```
#include <uchar.h>
        size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
size_t c32rtomb(char * [[ core::noalias, core::writethrough ]] s, char32_t c32,
        mbstate_t * [[ core::noalias ]] ps)
                [[ core::modifies(errno) ]]
```

Description

2 If `s` is a null pointer, the `c32rtomb` function is equivalent to the call

```
                c32rtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the `c32rtomb` function determines the number of bytes (including any shift sequences) needed to represent the multibyte character that corresponds to the ~~wide character~~ universal character name given by `c32` ~~(including any shift sequences)~~, and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most `MB_CUR_MAX` bytes are stored. If `c32` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The `c32rtomb` function returns the number of bytes ~~stored in the array object~~ (including any shift sequences) stored in the array object. When `c32` is not a valid ~~wide character~~ universal character name, an encoding error occurs: the function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)`(-1); the conversion state is unspecified.

7.29 Extended multibyte and wide character utilities `<wchar.h>`

7.29.1 Introduction

1 The header `<wchar.h>` defines four macros, and declares ~~four~~ three data types, one tag, and many functions.[433)]

2 The feature test macro `__CORE_VERSION_WCHAR_H__` expands to the token `202005L`.

3 The types declared are **wchar_t** ~~and **size_t** (both described in 7.19);~~

```
mbstate_t
```

which is a complete object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

```
wint_t
```

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **WEOF** below);[434)] and

```
struct tm
```

which is declared as an incomplete structure type (the contents are described in 7.27.1).

4 The macros defined are ~~**NULL** (described in 7.19);~~ **WCHAR_MIN**, **WCHAR_MAX**, and **WCHAR_WIDTH** (described in 7.20); and

```
WEOF
```

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set.[435)] It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

5 The functions declared are grouped as follows:

— Functions that perform input and output of wide characters, or multibyte characters, or both;

— Functions that provide wide string numeric conversion;

— Functions that perform general wide string manipulation;

— Functions for wide string date and time conversion; and

— Functions that provide extended capabilities for conversion between multibyte and wide character sequences.

6 Arguments to the functions in this subclause may point to arrays containing **wchar_t** values that do not correspond to members of the extended character set. Such values shall be processed according to the specified semantics, except that it is unspecified whether an encoding error occurs if such a value appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by **wcrtomb**.

7 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

433) See "future library directions" (7.32.15).
434) **wchar_t** and **wint_t** can be the same integer type.
435) The value of the macro **WEOF** can differ from that of **EOF** and need not be negative.

7.29.2 Formatted wide character input/output functions

1 The formatted wide character input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[436)]

7.29.2.1 The `fwprintf` function

Synopsis

1

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE * [[core::noalias]] stream, const wchar_t * [[core::noalias]]
    format, ...)
      [[core::modifies(errno)]];
```

Description

2 The `fwprintf` function writes output to the stream pointed to by `stream`, under control of the wide string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fwprintf` function returns when the end of the format string is encountered.

3 The format is composed of zero or more directives: ordinary wide characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the wide character `%`. After the `%`, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a nonnegative decimal integer.[437)]

— An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point wide character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of wide characters to be written for `s` conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag wide characters and their meanings are:

`-` The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

[436)] The `fwprintf` functions perform writes to memory for the `%n` specifier.

[437)] Note that `0` is taken as a flag, not as the beginning of a field width.

`+` The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)[438)]

space If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is prefixed to the result. If the *space* and `+` flags both appear, the *space* flag is ignored.

`#` The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`signed char`** or **`unsigned char`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`signed char`** or **`unsigned char`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`signed char`** argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`short int`** or **`unsigned short int`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`short int`** or **`unsigned short int`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`short int`** argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long int`** or **`unsigned long int`** argument; that a following `n` conversion specifier applies to a pointer to a **`long int`** argument; that a following `c` conversion specifier applies to a **`wint_t`** argument; that a following `s` conversion specifier applies to a pointer to a **`wchar_t`** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long long int`** or **`unsigned long long int`** argument; or that a following `n` conversion specifier applies to a pointer to a **`long long int`** argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **`intmax_t`** or **`uintmax_t`** argument; or that a following `n` conversion specifier applies to a pointer to an **`intmax_t`** argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`size_t`** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **`size_t`** argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`ptrdiff_t`** or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a **`ptrdiff_t`** argument.

438)The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`long double`** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

`d,i` The **`int`** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

`o,u,x,X` The **`unsigned int`** argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

`f,F` A **`double`** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **`#`** flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **`double`** argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A **`double`** argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-wchar-sequence*`)` — which style, and the meaning of any *n-wchar-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively.[439)]

`e,E` A **`double`** argument representing a floating-point number is converted in the style *[-]d.ddd*`e`*±dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point wide character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **`#`** flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g,G` A **`double`** argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of X:

if $P > X \geq -4$, the conversion is with style `f` (or `F`) and precision $P - (X + 1)$.

otherwise, the conversion is with style `e` (or `E`) and precision $P - 1$.

Finally, unless the **`#`** flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point wide character is removed if there is no fractional portion remaining.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

439)When applied to infinite and NaN values, the `-`, `+`, and *space* flag wide characters have their usual meaning; the **`#`** and `0` flag wide characters have no effect.

`a,A` A **double** argument representing a floating-point number is converted in the style *[-]*`0x`*h.hhhh*`p`$\pm$*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character[440] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish[441] values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` length modifier is present, the **int** argument is converted to a wide character as if by calling **btowc** and the resulting wide character is written.

If an `l` length modifier is present, the **wint_t** argument is converted to **wchar_t** and written.

`s` If no `l` length modifier is present, the argument shall be a pointer to the initial element of a character array containing a multibyte character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

`p` The argument shall be a pointer to **void**. The value of the pointer shall be valid or null. It is converted to a sequence of printing wide characters, in an implementation-defined manner. If the value of the pointer is valid its provenance is henceforth exposed.

`n` The argument shall be a pointer to signed integer into which is *written* the number of wide characters written to the output stream so far by this call to **fwprintf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%` A `%` wide character is written. No argument is converted. The complete conversion specification shall be `%%`.

9 If a conversion specification is invalid, the behavior is undefined.[442] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For `a` and `A` conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

---

[440] Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries.

[441] The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **FLT_RADIX** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point wide character.

[442] See "future library directions" (7.32.15).

Recommended practice

12 For `a` and `A` conversions, if **`FLT_RADIX`** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most the maximum value M of the *T*`_DECIMAL_DIG` macros (defined in `<float.h>`), then the result should be correctly rounded.[443] If the number of significant decimal digits is more than M but the source value is exactly representable with M digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having M significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

14 The **`fwprintf`** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

15 The number of wide characters that can be produced by any single conversion shall be at least 4095.

16 **EXAMPLE** To print a date and time in the form “Sunday, July 3, 10:02” followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
#include <wchar.h>
/* ... */
wchar_t *weekday, *month;  // pointers to wide strings
int day, hour, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));
fwprintf(stdout, L"pi = %.5f\n", 4 × atan(1.0));
```

Forward references: the **`btowc`** function (7.29.6.1.1), the **`mbrtowc`** function (7.29.6.3.2).

7.29.2.2 The `fwscanf` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE * [[ core::noalias ]]  stream, const wchar_t * [[ core::noalias ]]  format
    , ...)
      [[ core::modifies(errno) ]] ;
```

Description

2 The **`fwscanf`** function reads input from the stream pointed to by `stream`, under control of the wide string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format is composed of zero or more directives: one or more white-space wide characters, an ordinary wide character (neither `%` nor a white-space wide character), or a conversion specification. Each conversion specification is introduced by the wide character `%`. After the `%`, the following appear in sequence:

[443] For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

— An optional assignment-suppressing wide character `*`.

— An optional decimal integer greater than zero that specifies the maximum field width (in wide characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

4 The **`fwscanf`** function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read. The directive never fails.

6 A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If that wide character differs from the directive,the directive fails and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space wide characters are skipped, unless the specification includes a `[`, `c`, or `n` specifier.[444)]

9 An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input wide characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[445)] The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`signed char`** or **`unsigned char`**.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`short int`** or **`unsigned short int`**.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`long int`** or **`unsigned long int`**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`double`**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **`wchar_t`**.

`ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`long long int`** or **`unsigned long long int`**.

444) These white-space wide characters are not counted against a specified field width.

445) **`fwscanf`** pushes back at most one input wide character onto the input stream. Therefore, some sequences that are acceptable to **`wcstod`**, **`wcstol`**, etc., are unacceptable to **`fwscanf`**.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`intmax_t`** or **`uintmax_t`**.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`size_t`** or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`ptrdiff_t`** or the corresponding unsigned integer type.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`long double`**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **`wcstol`** function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to signed integer.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **`wcstol`** function with the value 0 for the `base` argument. The corresponding argument shall be a pointer to signed integer.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **`wcstoul`** function with the value 8 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **`wcstoul`** function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **`wcstoul`** function with the value 16 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`a,e,f,g` Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **`wcstod`** function. The corresponding argument shall be a pointer to floating.

`c` Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the directive).

If no `l` length modifier is present, characters from the input field are converted as if by repeated calls to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the sequence.No null wide character is added.

`s` Matches a sequence of non-white-space wide characters.

If no `l` length modifier is present, characters from the input field are converted as if by repeated calls to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

`[` Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*).

If no `l` length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the `format` string, up to and including the matching right bracket (`]`). The wide characters between the brackets (the *scanlist*) compose the scanset, unless the wide character after the left bracket is a circumflex (`^`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket wide character is in the scanlist and the next following right bracket wide character is the matching right bracket that ends the specification; otherwise the first following right bracket wide character is the one that ends the specification. If a `-` wide character is in the scanlist and is not the first, nor the second where the first wide character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches the same implementation-defined set of sequences of wide characters that may be produced by the `%p` conversion of the **fwprintf** function. The corresponding argument `ptr` shall be a pointer to a pointer to **void**.

- If the input sequence could have been printed from a null pointer value, `*ptr` is assigned a null pointer value.
- Otherwise, if the input sequence could have been printed from a valid pointer x and if the address x currently refers to an exposed storage instance, a valid pointer with address x and the provenance of that storage instance is synthesized in `*ptr`.[446)]
- Otherwise `*ptr` becomes indeterminate.

`n` No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of wide characters read from the input stream so far by this call to the **fwscanf** function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the **fwscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

`%` Matches a single `%` wide character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[447)]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

446)Thus, the constructed pointer value has a valid provenance. Nevertheless, because the original storage instance might be dead and a new storage instance might live at the same address, this provenance can be different from the provenance that gave rise to the print operation.If x can be an address with more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

447)See "future library directions" (7.32.15).

15 Trailing white-space wide characters(including new-line wide characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns

16 The **`fwscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 **EXAMPLE 1** The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int n, i; float x; wchar_t name[50];
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `thompson\0`.

18 **EXAMPLE 2** The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int i; float x; double y;
fwscanf(stdin, L"%2d%f%*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip past `0123`, and will assign to `y` the value 56.0. The next wide character read from the input stream will be `a`.

Forward references: the ~~**`wcstod`**, **`wcstof`**, and **`wcstold`** functions (??~~**`strtod`**, **`strtof`**, and **`strtold`** type-generic macros (7.22.1.3), the ~~**`wcstol`**, **`wcstoll`**, **`wcstoul`**, and **`wcstoull`** functions (??~~**`strtol`**, **`strtoll`**, **`strtoul`**, and **`strtoull`** type-generic macros (7.22.1.4), the **`wcrtomb`** function (7.29.6.3.3).

7.29.2.3 The `swprintf` function

Synopsis

1
```
#include <wchar.h>
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
     ...);

int swprintf(wchar_t * [[ core::noalias ]] s, size_t n, const wchar_t * [[ core::noalias ]] format,
     ...) [[ core::modifies(errno) ]] ;
```

Description

2 The **`swprintf`** function is equivalent to **`fwprintf`**, except that the argument `s` specifies an array of wide characters into which the generated output is to be written, rather than written to a stream. No more than `n` wide characters are written, including a terminating null wide character, which is always added (unless `n` is zero).

Returns

3 The **`swprintf`** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be written.

7.29.2.4 The swscanf function

Synopsis

1
```
#include <wchar.h>
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);

int swscanf(const wchar_t *[[core::noalias]] s, const wchar_t *[[core::noalias]] format, ...)
        [[core::modifies(errno)]];
```

Description

2 The **swscanf** function is equivalent to **fwscanf**, except that the argument s specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf** function.

Returns

3 The **swscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **swscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.5 The vfwprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,
        va_list arg);

int vfwprintf(FILE *[[core::noalias]] stream, const wchar_t *[[core::noalias]] format,
        va_list arg) [[core::modifies(errno)]];
```

Description

2 The **vfwprintf** function is equivalent to **fwprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwprintf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vfwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

4 **EXAMPLE** The following shows the use of the **vfwprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

void error(char *function_name, wchar_t *format, ...)
void error(char *function_name, wchar_t *format, ...)
{
        va_list args;

        va_start(args, format);
        // print out name of function causing error
        fwprintf(stderr, L"ERROR in %s: ", function_name);
        // print out remainder of message
        vfwprintf(stderr, format, args);
```

448) As the functions **vfwprintf**, **vswprintf**, **vfwscanf**, **vwprintf**, **vwscanf**, and **vswscanf** invoke the **va_arg** macro, the value of arg after the return is indeterminate.

```
        va_end(args);
    }
```

7.29.2.6 The `vfwscanf` function

Synopsis

1
```
    #include <stdarg.h>
    #include <stdio.h>
    #include <wchar.h>
    int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
            va_list arg);

    int vfwscanf(FILE * [[ core::noalias ]] stream, const wchar_t * [[ core::noalias ]] format,
            va_list arg) [[ core::modifies(errno) ]] ;
```

Description

2 The **vfwscanf** function is equivalent to **fwscanf**, with the variable argument list replaced by `arg`, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfwscanf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vfwscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfwscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.7 The `vswprintf` function

Synopsis

1
```
    #include <stdarg.h>
    #include <wchar.h>
    int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
            va_list arg);

    int vswprintf(wchar_t * [[ core::noalias ]] s, size_t n, const wchar_t * [[ core::noalias ]] format,
            va_list arg) [[ core::modifies(errno) ]] ;
```

Description

2 The **vswprintf** function is equivalent to **swprintf**, with the variable argument list replaced by `arg`, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswprintf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vswprintf** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be generated.

7.29.2.8 The `vswscanf` function

Synopsis

1
```
    #include <stdarg.h>
    #include <wchar.h>
    int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,
            va_list arg);

    int vswscanf(const wchar_t * [[ core::noalias ]] s, const wchar_t * [[ core::noalias ]] format,
            va_list arg) [[ core::modifies(errno) ]] ;
```

Description

2 The **vswscanf** function is equivalent to **swscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswscanf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vswscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vswscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.9 The vwprintf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
~~int vwprintf(const wchar_t * restrict format, va_list arg);~~
int vwprintf(const wchar_t * [[ core::noalias ]] format, va_list arg)
        [[ core::modifies(errno), core::evaluates(stdout) ]] ;
```

Description

2 The **vwprintf** function is equivalent to **wprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vwprintf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.10 The vwscanf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
~~int vwscanf(const wchar_t * restrict format, va_list arg);~~
int vwscanf(const wchar_t * [[ core::noalias ]] format, va_list arg)
        [[ core::modifies(errno), core::evaluates(stdin) ]] ;
```

Description

2 The **vwscanf** function is equivalent to **wscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vwscanf** function does not invoke the **va_end** macro.[448)]

Returns

3 The **vwscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vwscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.11 The wprintf function

Synopsis

1
```
#include <wchar.h>
~~int wprintf(const wchar_t * restrict format, ...);~~
int wprintf(const wchar_t * [[ core::noalias ]] format, ...)
        [[ core::modifies(errno), core::evaluates(stdout) ]] ;
```

Description

2 The **wprintf** function is equivalent to **fwprintf** with the argument **stdout** interposed before the arguments to **wprintf**.

Returns

3 The **wprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.12 The wscanf function

Synopsis

1
```
#include <wchar.h>
int wscanf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * [[core::noalias]] format, ...)
        [[core::modifies(errno), core::evaluates(stdin)]];
```

Description

2 The **wscanf** function is equivalent to **fwscanf** with the argument **stdin** interposed before the arguments to **wscanf**.

Returns

3 The **wscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **wscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.3 Wide character input/output functions

7.29.3.1 The fgetwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
wint_t fgetwc(FILE *stream) [[core::modifies(errno)]];
```

Description

2 If the end-of-file indicator for the input stream pointed to by stream is not set and a next wide character is present, the **fgetwc** function obtains that wide character as a **wchar_t** converted to a **wint_t** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the **fgetwc** function returns **WEOF**. Otherwise, the **fgetwc** function returns the next wide character from the input stream pointed to by stream. If a read error occurs, the error indicator for the stream is set and the **fgetwc** function returns **WEOF**. If an encoding error occurs (including too few bytes), the error indicator for the stream is set and the value of the macro **EILSEQ** is stored in **errno** and the **fgetwc** function returns **WEOF**.[449]

7.29.3.2 The fgetws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

[449] An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions. Also, **errno** will be set to **EILSEQ** by input/output functions only if an encoding error occurs.

```
wchar_t *fgetws(wchar_t * [[ core::noalias ]] s, int n, FILE * [[ core::noalias ]] stream)
          [[ core::modifies(errno) ]];
```

Description

2 The **fgetws** function reads at most one less than the number of wide characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

Returns

3 The **fgetws** function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

7.29.3.3 The fputwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
~~wint_t fputwc(wchar_t c, FILE *stream);~~
wint_t fputwc(wchar_t c, FILE *stream) [[ core::modifies(errno) ]];
```

Description

2 The **fputwc** function writes the wide character specified by `c` to the output stream pointed to by `stream`, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **fputwc** function returns the wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc** returns **WEOF**. If an encoding error occurs, the value of the macro **EILSEQ** is stored in **errno** and **fputwc** returns **WEOF**.

7.29.3.4 The fputws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
~~int fputws(const wchar_t * restrict s, FILE * restrict stream);~~
int fputws(const wchar_t * [[ core::noalias ]] s, FILE * [[ core::noalias ]] stream)
          [[ core::modifies(errno) ]];
```

Description

2 The **fputws** function writes the wide string pointed to by `s` to the stream pointed to by `stream`. The terminating null wide character is not written.

Returns

3 The **fputws** function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

7.29.3.5 The fwide function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

2 The **fwide** function determines the orientation of the stream pointed to by `stream`. If `mode` is greater than zero, the function first attempts to make the stream wide oriented. If `mode` is less than zero, the function first attempts to make the stream byte oriented.[450] Otherwise, `mode` is zero and the function does not alter the orientation of the stream.

Returns

3 The **fwide** function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

7.29.3.6 The getwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

2 The **getwc** function is equivalent to **fgetwc**, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects.

Returns

3 The **getwc** function returns the next wide character from the input stream pointed to by `stream`, or **WEOF**.

7.29.3.7 The getwchar function

Synopsis

1
```
#include <wchar.h>
wint_t getwchar(void);
wint_t getwchar(void) [[core::evaluates(stdin)]];
```

Description

2 The **getwchar** function is equivalent to **getwc** with the argument **stdin**.

Returns

3 The **getwchar** function returns the next wide character from the input stream pointed to by **stdin**, or **WEOF**.

7.29.3.8 The putwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

Description

2 The **putwc** function is equivalent to **fputwc**, except that if it is implemented as a macro, it may evaluate `stream` more than once, so that argument should never be an expression with side effects.

Returns

3 The **putwc** function returns the wide character written, or **WEOF**.

7.29.3.9 The putwchar function

[450] If the orientation of the stream has already been determined, **fwide** does not change it.

Synopsis

1
```
#include <wchar.h>
wint_t putwchar(wchar_t c);
wint_t putwchar(wchar_t c)  [[core::evaluates(stdout)]];;
```

Description

2 The `putwchar` function is equivalent to `putwc` with the second argument `stdout`.

Returns

3 The `putwchar` function returns the character written, or `WEOF`.

7.29.3.10 The `ungetwc` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

2 The `ungetwc` function pushes the wide character specified by `c` back onto the input stream pointed to by `stream`. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file positioning function (`fseek`, `fsetpos`, or `rewind`) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

3 One wide character of pushback is guaranteed, even if the call to the `ungetwc` function follows just after a call to a formatted wide character input function `fwscanf`, `vfwscanf`, `vwscanf`, or `wscanf`. If the `ungetwc` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of `c` equals that of the macro `WEOF`, the operation fails and the input stream is unchanged.

5 A successful call to the `ungetwc` function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.[451] For a text or binary stream, the value of its file position indicator after a successful call to the `ungetwc` function is unspecified until all pushed-back wide characters are read or discarded.

Returns

6 The `ungetwc` function returns the wide character pushed back, or `WEOF` if the operation fails.

7.29.4 General utilities for wide character arrays

1 The header `<wchar.h>` declares a number of functions useful for ~~wide string manipulation . Various methods are used for determining the lengths of the arrays, but in~~ manipulation wide character arrays. In all cases a `wchar_t*` argument points to the initial (lowest addressed) element of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 ~~Where an argument~~ An argument is declared as `size_t n` that determines the length of the array for a function~~, ;~~ `n` can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters. ~~Synopsis replace~~

~~**Description**~~

~~The `wcstod`, `wcstof`, and `wcstold` functions convert the initial portion of the wide string pointed to by `nptr` to `double`, `float`, and `long double` representation, respectively. First, they decompose the~~

[451]Note that a file positioning function could further modify the file position indicator after discarding any pushed-back wide characters.

~~input string into three parts: an initial, possibly empty, sequence of white-space wide characters, a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.~~

3 ~~The expected form of the subject sequence is an optional plus or minus sign, then one of the following: a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in 6.4.4.2; a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part as defined in 6.4.4.2; `INF` or `INFINITY`, or any other wide string equivalent except for case `NAN` or `NAN(`*n-wchar-sequence*$_{opt}$`)`, or any other wide string equivalent except for case in the `NAN` part, where:~~

~~*n-wchar-sequence:*~~
~~*digit*~~
~~*nondigit*~~
~~*n-wchar-sequence digit*~~
~~*n-wchar-sequence nondigit* The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.~~

~~If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated.A wide character sequence `INF` or `INFINITY` is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence `NAN` or `NAN(`*n-wchar-sequence*$_{opt}$`)` is interpreted as a quiet NaN, if supported in the return type , else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequence is implementation-defined~~Previous versions of this standard declared functions to handle wide-character strings of base type **`wchar_t`** . ~~A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.~~

~~If the subject sequence has the hexadecimal form and is a power of 2, the value resulting from the conversion is correctly rounded.~~

~~In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.~~

~~If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.~~

~~**Recommended practice**~~

~~If the subject sequence has the hexadecimal form, **`FLT_RADIX`** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.~~

~~If the subject sequence has the decimal form and at most M significant digits, where M is the maximum value of the *T*__**`DECIMAL_DIG`** macros (defined in), the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than M significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having M significant digits, such that the values of *L*, *D*, and *U* satisfy *L D U*. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding~~

~~direction~~These functions are obsolecent and may be replaced by the corresponding type-generic macros from the `<string.h>` header.

~~**Returns**~~

~~The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus , , or is returned (according to the return type and sign of the value), and the value of the macro is stored in . If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether **errno** acquires the value **ERANGE** is implementation-defined.~~

~~Synopsis replace~~
~~**Description**~~

~~The **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions convert the initial portion of the wide string pointed to by `nptr` to **long int**, **long long int**, **unsigned long int**, and **unsigned long long int** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters, a subject sequence resembling an integer represented in some radix determined by the value of `base`, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.~~

~~If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide characters `0x` or `0X` may optionally precede the sequence of letters and digits, followingthe sign if present.~~

~~The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white-space wide characters, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.~~

~~If the subject sequence has the expected form and the value of `base` is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.~~

~~In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.~~

~~If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.~~

~~**Returns**~~The names of the obsolecent functions are still reserved. They are the following:

wcscat	**wcscspn**	**wcspbrk**	**wcstof**	**wcstoul**
wcschr	**wcslen**	**wcsrchr**	**wcstok**	**wcstoull**
wcscmp	**wcsncat**	**wcsspn**	**wcstol**	**wcstoumax**
wcscoll	**wcsncmp**	**wcsstr**	**wcstold**	**wcsxfrm**
wcscpy	**wcsncpy**	**wcstod**	**wcstoll**	

~~The wcstol, wcstoll, wcstoul, and wcstoull functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, , , , , , or is returned (according to the return type sign of the value, if any), and the value of the macro is stored in .~~

7.29.4.1 Wide string copying functions

~~Synopsis replace~~

~~**Description**~~

~~The copies the wide string pointed to by s2 (including the terminating null wide character) into the array pointed to by s1.~~

~~**Returns**~~

~~The returns the value of s1.~~

~~Synopsis replace~~

~~**Description**~~

~~The copies not more than n wide characters (those that follow a null wide character are not copied) from the array pointed to by s2 to the array pointed to by s1.~~

~~If the array pointed to by s2 is a wide string that is shorter than n wide characters, null wide characters are appended to the copy in the array pointed to by s1, until n wide characters in all have been written.~~

~~**Returns**~~

~~The returns the value of s1.~~

7.29.4.1.1 The wmemcpy function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemcpy(wchar_t * [[ core::noalias ]] s1, const wchar_t * [[ core::noalias ]] s2, size_t n);
```

Description

2 The wmemcpy function copies n wide characters from the object pointed to by s2 to the object pointed to by s1.

Returns

3 The wmemcpy function returns the value of s1.

7.29.4.1.2 The wmemmove function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

2 The wmemmove function copies n wide characters from the object pointed to by s2 to the object pointed to by s1. Copying takes place as if the n wide characters from the object pointed to by s2 are first copied into a temporary array of n wide characters that does not overlap the objects pointed to by s1 or s2, and then the n wide characters from the temporary array are copied into the object pointed to by s1.

Returns

3 The wmemmove function returns the value of s1.

~~Synopsis replace~~

~~**Description**~~

~~The appends a copy of the wide string pointed to by s2 (including the terminating null wide character) to the end of the wide string pointed to by s1. The initial wide character of s2 overwrites the null wide character at the end of s1.~~

~~**Returns**~~

~~The returns the value of s1.~~

~~Synopsis replace~~
~~**Description**~~

~~The appends not more than n wide characters (a null wide character and those that follow it are not appended) from the array pointed to by s2 to the end of the wide string pointed to by s1. The initial wide character of s2 overwrites the null wide character at the end of s1. A terminating null wide character is always appended to the result.~~

~~**Returns**~~

~~The returns the value of s1.~~

7.29.4.2 Wide string comparison functions

1 Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integer type designated by **wchar_t**.

~~Synopsis replace~~
~~**Description**~~

~~The compares the wide string pointed to by s1 to the wide string pointed to by s2.~~

~~**Returns**~~

~~The returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by s1 is greater than, equal to, or less than the wide string pointed to by s2.~~

~~Synopsis replace~~
~~**Description**~~

~~The compares the wide string pointed to by s1 to the wide string pointed to by s2, both interpreted as appropriate to the category of the current locale.~~

~~**Returns**~~

~~The returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by s1 is greater than, equal to, or less than the wide string pointed to by s2 when both are interpreted as appropriate to the current locale.~~

~~Synopsis replace~~
~~**Description**~~

~~The compares not more than n wide characters (those that follow a null wide character are not compared) from the array pointed to by s1 to the array pointed to by s2.~~

~~**Returns**~~

~~The returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2.~~

~~Synopsis replace~~
~~**Description**~~

~~The transforms the wide string pointed to by s2 and places the resulting wide string into the array pointed to by s1. The transformation is such that if the function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the function applied to the same two original wide strings. No more than n wide characters are placed into the resulting array pointed to by s1, including the terminating null wide character. If n is zero, s1 is permitted to be a null pointer.~~

~~**Returns**~~

~~The returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is `n` or greater, the contents of the array pointed to by `s1` are indeterminate. The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by `s`:~~

7.29.4.2.1 The `wmemcmp` function

Synopsis

1
```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

2 The `wmemcmp` function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`.

Returns

3 The `wmemcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

7.29.4.3 Wide string search functions

~~Synopsis replace~~
~~**Description**~~

~~The locates the first occurrence of `c` in the wide string pointed to by `s`. The terminating null wide character is considered to be part of the wide string.~~

~~**Returns**~~

~~The returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.~~

~~Synopsis replace~~
~~**Description**~~

~~The computes the length of the maximum initial segment of the wide string pointed to by `s1` which consists entirely of wide characters *not* from the wide string pointed to by `s2`.~~

~~**Returns**~~

~~The returns the length of the segment.~~

~~Synopsis replace~~
~~**Description**~~

~~The locates the first occurrence in the wide string pointed to by `s1` of any wide character from the wide string pointed to by `s2`.~~

~~**Returns**~~

~~The returns a pointer to the wide character in `s1`, or a null pointer if no wide character from `s2` occurs in `s1`.~~

~~Synopsis replace~~
~~**Description**~~

~~The locates the last occurrence of `c` in the wide string pointed to by `s`. The terminating null wide character is considered to be part of the wide string.~~

~~**Returns**~~

~~The returns a pointer to the wide character, or a null pointer if `c` does not occur in the wide string.~~

~~Synopsis replace~~
~~**Description**~~

~~The computes the length of the maximum initial segment of the wide string pointed to by `s1` which consists entirely of wide characters from the wide string pointed to by `s2`.~~

~~**Returns**~~

~~The returns the length of the segment.~~

~~Synopsis replace~~
~~**Description**~~

~~The locates the first occurrence in the wide string pointed to by `s1` of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by `s2`.~~

~~**Returns**~~

~~The returns a pointer to the located wide string, or a null pointer if the wide string is not found. If `s2` points to a wide string with zero length, the function returns `s1`.~~

~~Synopsis replace~~
~~**Description**~~

~~A sequence of calls to the breaks the wide string pointed to by `s1` into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by `s2`. The third argument points to a caller-provided **`wchar_t`** pointer into which the stores information necessary for it to continue scanning the same wide string.~~

~~The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by `ptr`. Subsequent calls in the sequence have a null first argument and the object pointed to by `ptr` is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by `s2` may be different from call to call.~~

~~The first call in the sequence searches the wide string pointed to by `s1` for the first wide character that is *not* contained in the current separator wide string pointed to by `s2`. If no such wide character is found, then there are no tokens in the wide string pointed to by `s1` and the returns a null pointer. If such a wide character is found, it is the start of the first token.~~

~~The then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by `s1`, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.~~

~~In all cases, the stores sufficient information in the pointer pointed to by `ptr` so that subsequent calls, with a null pointer for `s1` and the unmodified pointer value for `ptr`, shall start searching just past the element overwritten by a null wide character (if any).~~

~~**Returns**~~

~~The returns a pointer to the first wide character of a token, or a null pointer if there is no token.~~

7.29.4.3.1 The `wmemchr` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

2 The `wmemchr` function locates the first occurrence of `c` in the initial `n` wide characters of the object pointed to by `s`.

Returns

3 The `wmemchr` function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

7.29.4.4 Miscellaneous functions

~~Synopsis replace~~

~~**Description**~~

~~The computes the length of the wide string pointed to by s.~~

~~**Returns**~~

~~The returns the number of wide characters that precede the terminating null wide character.~~

7.29.4.4.1 The wmemset function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

2 The wmemset function copies the value of c into each of the first n wide characters of the object pointed to by s.

Returns

3 The wmemset function returns the value of s.

7.29.5 Wide character time conversion functions

7.29.5.1 The wcsftime function

Synopsis

1
```
#include <time.h>
#include <wchar.h>
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
        const wchar_t * restrict format, const struct tm * restrict timeptr);
size_t wcsftime(wchar_t * [[core::noalias]] s, size_t maxsize,
    const wchar_t * [[core::noalias]] format, const struct tm * [[core::noalias]] timeptr)
        [[core::evaluates(locale)]] ;
```

Description

2 The wcsftime function is equivalent to the strftime function, except that:

— The argument s points to the initial element of an array of wide characters into which the generated output is to be placed.

— The argument maxsize indicates the limiting number of wide characters.

— The argument format is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.

— The return value indicates the number of wide characters.

Returns

3 If the total number of resulting wide characters including the terminating null wide character is not more than maxsize, the wcsftime function returns the number of wide characters placed into the array pointed to by s not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

7.29.6 Extended multibyte/wide character conversion utilities

1 The header <wchar.h> declares an extended set of functions useful for conversion between multibyte characters and wide characters.

2 Most of the following functions — those that are listed as “restartable”, 7.29.6.3 and 7.29.6.4 — take as a last argument a pointer to an object of type mbstate_t that is used to describe the current

conversion state from a particular multibyte character sequence to a wide character sequence (or the reverse) under the rules of a particular setting for the `LC_CTYPE` category of the current locale.

3 The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new multibyte character in the initial shift state. A zero-valued `mbstate_t` object is (at least) one way to describe an initial conversion state. A zero-valued `mbstate_t` object can be used to initiate conversion involving any multibyte character sequence, in any `LC_CTYPE` category setting. If an `mbstate_t` object has been altered by any of the functions described in this subclause, and is then used with a different multibyte character sequence, or in the other conversion direction, or with a different `LC_CTYPE` category setting than on earlier function calls, the behavior is undefined.[452]

4 On entry, each function takes the described conversion state (either internal or pointed to by an argument) as current. The conversion state described by the referenced object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

5 The behavior of the functions in this subclause is affected by the `LC_CTYPE` category of the current locale. An atribute corresponding to the pragma

```
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(locale)
```

is implied for the remainder of the functions provided by this header.

7.29.6.1 Single-byte/wide character conversion functions

7.29.6.1.1 The `btowc` function

Synopsis

1
```
#include <wchar.h>
wint_t btowc(int c);
```

Description

2 The `btowc` function determines whether `c` constitutes a valid single-byte character in the initial shift state.

Returns

3 The `btowc` function returns `WEOF` if `c` has the value `EOF` or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state. Otherwise, it returns the wide character representation of that character.

7.29.6.1.2 The `wctob` function

Synopsis

1
```
#include <wchar.h>
int wctob(wint_t c);
```

Description

2 The `wctob` function determines whether `c` corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Returns

3 The `wctob` function returns `EOF` if `c` does not correspond to a multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an `unsigned char` converted to an `int`.

7.29.6.2 Conversion state functions

7.29.6.2.1 The `mbsinit` function

[452] Thus, a particular `mbstate_t` object can be used, for example, with both the `mbrtowc` and `mbsrtowcs` functions as long as they are used to step sequentially through the same multibyte character string.

Synopsis

1
```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

2 If ps is not a null pointer, the **mbsinit** function determines whether the referenced **mbstate_t** object describes an initial conversion state.

Returns

3 The **mbsinit** function returns nonzero if ps is a null pointer or if the referenced object describes an initial conversion state; otherwise, it returns zero.

7.29.6.3 Restartable multibyte/wide character conversion functions

1 These functions differ from the corresponding multibyte character functions of 7.22.8 (**mblen**, **mbtowc**, and **wctomb**) in that they have an extra parameter, ps, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If ps is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for ps.

2 Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.29.6.3.1 The mbrlen function

Synopsis

1
```
#include <wchar.h>
~~size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);~~
size_t mbrlen(const char * [[core::noalias]] s, size_t n, mbstate_t * [[core::noalias]] ps);
```

Description

2 The **mbrlen** function is equivalent to the call:

```
~~mbrtowc(NULL, s, n, ps != NULL ? ps: &internal)~~
mbrtowc(nullptr, s, n, ps ≠ nullptr ? ps : &internal)
```

where internal is the **mbstate_t** object for the **mbrlen** function, except that the expression designated by ps is evaluated only once.

Returns

3 The **mbrlen** function returns a value between zero and n, inclusive, **(size_t)** (-2), or **(size_t)** (-1).

Forward references: the **mbrtowc** function (7.29.6.3.2).

7.29.6.3.2 The mbrtowc function

Synopsis

1
```
#include <wchar.h>
~~size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n,~~
~~        mbstate_t * restrict ps);~~
size_t mbrtowc(wchar_t * [[core::noalias]] pwc, const char * [[core::noalias]] s, size_t n,
        mbstate_t * [[core::noalias]] ps) [[core::modifies(errno)]];
```

Description

2 If s is a null pointer, the **mbrtowc** function is equivalent to the call:

```
~~mbrtowc(NULL, "", 1, ps)~~
mbrtowc(nullptr, "", 1, ps)
```

In this case, the values of the parameters `pwc` and `n` are ignored.

3 If `s` is not a null pointer, the **`mbrtowc`** function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **`mbrtowc`** function returns the first of the following that applies (given the current conversion state):

0 — if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* — if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)`(-2) — if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[453]

`(size_t)(-1)` — if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **`EILSEQ`** is stored in **`errno`**, and the conversion state is unspecified.

7.29.6.3.3 The `wcrtomb` function

Synopsis

1
```
#include <wchar.h>
~~size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);~~
size_t wcrtomb(char * [[core::noalias]] s, wchar_t wc, mbstate_t * [[core::noalias]] ps)
        [[core::modifies(errno)]];
```

Description

2 If `s` is a null pointer, the **`wcrtomb`** function is equivalent to the call

```
wcrtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the **`wcrtomb`** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most **`MB_CUR_MAX`** bytes are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **`wcrtomb`** function returns the number of bytes stored in the array object (including any shift sequences). When `wc` is not a valid wide character, an encoding error occurs: the function stores the value of the macro **`EILSEQ`** in **`errno`** and returns `(size_t)`(-1); the conversion state is unspecified.

[453] When `n` has at least the value of the **`MB_CUR_MAX`** macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.29.6.4 Restartable multibyte/wide string conversion functions

1 These functions differ from the corresponding multibyte string functions of 7.22.9 (**mbstowcs** and **wcstombs**) in that they have an extra parameter, `ps`, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

2 Also unlike their corresponding functions, the conversion source parameter, `src`, has a pointer-to-pointer type. When the function is storing the results of conversions (that is, when `dst` is not a null pointer), the pointer object pointed to by this parameter is updated to reflect the amount of the source processed by that invocation.

7.29.6.4.1 The mbsrtowcs function

Synopsis

1
```
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len,
          mbstate_t * restrict ps);

    size_t mbsrtowcs(wchar_t * [[core::noalias]] dst, const char ** [[core::noalias]] src, size_t len,
          mbstate_t * [[core::noalias]] ps) [[core::modifies(errno)]];
```

Description

2 The **mbsrtowcs** function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by `ps`, from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters are stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.[454] Each conversion takes place as if by a call to the **mbrtowc** function.

3 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if `dst` is not a null pointer, the resulting state described is the initial conversion state.

Returns

4 If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs** function stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null character (if any).

7.29.6.4.2 The wcsrtombs function

Synopsis

1
```
#include <wchar.h>
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len,
          mbstate_t * restrict ps);

    size_t wcsrtombs(char * [[core::noalias]] dst, const wchar_t ** [[core::noalias]] src, size_t len,
          mbstate_t * [[core::noalias]] ps) [[core::modifies(errno)]];
```

454) Thus, the value of `len` is ignored if `dst` is a null pointer.

Description

2 The `wcsrtombs` function converts a sequence of wide characters from the array indirectly pointed to by `src` into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters are then stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multibyte character, or (if `dst` is not a null pointer) when the next multibyte character would exceed the limit of `len` total bytes to be stored into the array pointed to by `dst`. Each conversion takes place as if by a call to the `wcrtomb` function.[455)]

3 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Returns

4 If conversion stops because a wide character is reached that does not correspond to a valid multibyte character, an encoding error occurs: the `wcsrtombs` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)`(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

[455)] If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

7.30 Wide character classification and mapping utilities `<wctype.h>`

7.30.1 Introduction

1 The header `<wctype.h>` defines one macro, and declares three data types and many functions.[456)]

2 The types declared are **`wint_t`** described in 7.29.1;

```
wctrans_t
```

which is a scalar type that can hold values which represent locale-specific character mappings; and

```
wctype_t
```

which is a scalar type that can hold values which represent locale-specific character classifications.

3 The macro defined is **`WEOF`** (described in 7.29.1).

4 The functions declared are grouped as follows:

- Functions that provide wide character classification;
- Extensible functions that provide wide character classification;
- Functions that provide wide character case mapping;
- Extensible functions that provide wide character mapping.

5 For all functions described in this subclause that accept an argument of type **`wint_t`**, the value shall be representable as a **`wchar_t`** or shall equal the value of the macro **`WEOF`**. If this argument has any other value, the behavior is undefined.

6 The behavior of these functions is affected by the **`LC_CTYPE`** category of the current locale. Atributes corresponding to the pragma

```
#pragma CORE_FUNCTION_ATTRIBUTE core::evaluates(locale)
```

are implied for the remainder of the functions provided by this header.

7.30.2 Wide character classification utilities

1 The header `<wctype.h>` declares several functions useful for classifying wide characters.

2 The term *printing wide character* refers to a member of a locale-specific set of wide characters, each of which occupies at least one printing position on a display device. The term *control wide character* refers to a member of a locale-specific set of wide characters that are not printing wide characters.

7.30.2.1 Wide character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument `wc` conforms to that in the description of the function.

2 Each of the following functions returns true for each wide character that corresponds (as if by a call to the **`wctob`** function) to a single-byte character for which the corresponding character classification function from 7.4.1 returns true, except that the **`iswgraph`** and **`iswpunct`** functions may differ with respect to wide characters other than `L' '` that are both printing and white-space wide characters.[457)]

Forward references: the **`wctob`** function (7.29.6.1.2).

[456)] See "future library directions" (7.32.16).

[457)] For example, if the expression **`isalpha`**`(`**`wctob`**`(wc))` evaluates to true, then the call **`iswalpha`**`(wc)` also returns true. But, if the expression **`isgraph`**`(`**`wctob`**`(wc))` evaluates to true (which cannot occur for `wc` ≡ `L' '` of course), then either **`iswgraph`**`(wc)` or **`iswprint`**`(wc)∧`**`iswspace`**`(wc)` is true, but not both.

7.30.2.1.1 The `iswalnum` function

Synopsis

1
```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

2 The **`iswalnum`** function tests for any wide character for which **`iswalpha`** or **`iswdigit`** is true.

7.30.2.1.2 The `iswalpha` function

Synopsis

1
```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

2 The **`iswalpha`** function tests for any wide character for which **`iswupper`** or **`iswlower`** is true, or any wide character that is one of a locale-specific set of alphabetic wide characters for which none of **`iswcntrl`**, **`iswdigit`**, **`iswpunct`**, or **`iswspace`** is true.[458)]

7.30.2.1.3 The `iswblank` function

Synopsis

1
```
#include <wctype.h>
int iswblank(wint_t wc);
```

Description

2 The **`iswblank`** function tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which **`iswspace`** is true and that is used to separate words within a line of text. The standard blank wide characters are the following: space (`L' '`), and horizontal tab (`L'\t'`). In the `"C"` locale, **`iswblank`** returns true only for the standard blank characters.

7.30.2.1.4 The `iswcntrl` function

Synopsis

1
```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

2 The **`iswcntrl`** function tests for any control wide character.

7.30.2.1.5 The `iswdigit` function

Synopsis

1
```
#include <wctype.h>
int iswdigit(wint_t wc);
constexpr int iswdigit(wint_t wc);
```

Description

2 The **`iswdigit`** function tests for any wide character that corresponds to a decimal-digit character (as defined in 5.2.1).

458)The functions **`iswlower`** and **`iswupper`** test true or false separately for each of these additional wide characters; all four combinations are possible.

7.30.2.1.6 The `iswgraph` function

Synopsis

1
```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

2 The **`iswgraph`** function tests for any wide character for which **`iswprint`** is true and **`iswspace`** is false.[459)]

7.30.2.1.7 The `iswlower` function

Synopsis

1
```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

2 The **`iswlower`** function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **`iswcntrl`**, **`iswdigit`**, **`iswpunct`**, or **`iswspace`** is true.

7.30.2.1.8 The `iswprint` function

Synopsis

1
```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

2 The **`iswprint`** function tests for any printing wide character.

7.30.2.1.9 The `iswpunct` function

Synopsis

1
```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

2 The **`iswpunct`** function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither **`iswspace`** nor **`iswalnum`** is true.[459)]

7.30.2.1.10 The `iswspace` function

Synopsis

1
```
#include <wctype.h>
int iswspace(wint_t wc);
```

Description

2 The **`iswspace`** function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **`iswalnum`**, **`iswgraph`**, or **`iswpunct`** is true.

7.30.2.1.11 The `iswupper` function

Synopsis

1
```
#include <wctype.h>
int iswupper(wint_t wc);
```

[459)]Note that the behavior of the **`iswgraph`** and **`iswpunct`** functions can differ from their corresponding functions in 7.4.1 with respect to printing, white-space, single-byte execution characters other than `' '`.

Description

2 The **iswupper** function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

7.30.2.1.12 The iswxdigit function

Synopsis

1
```
#include <wctype.h>
int iswxdigit(wint_t wc);
constexpr int iswxdigit(wint_t wc);
```

Description

2 The **iswxdigit** function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in 6.4.4.1).

7.30.2.2 Extensible wide character classification functions

1 The functions **wctype** and **iswctype** provide extensible wide character classification as well as testing equivalent to that performed by the functions described in the previous subclause (7.30.2.1).

7.30.2.2.1 The iswctype function

Synopsis

1
```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description

2 The **iswctype** function determines whether the wide character `wc` has the property described by `desc`. The current setting of the **LC_CTYPE** category shall be the same as during the call to **wctype** that returned the value `desc`.

3 Each of the following expressions has a truth-value equivalent to the call to the wide character classification function (7.30.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum"))    // iswalnum(wc)
iswctype(wc, wctype("alpha"))    // iswalpha(wc)
iswctype(wc, wctype("blank"))    // iswblank(wc)
iswctype(wc, wctype("cntrl"))    // iswcntrl(wc)
iswctype(wc, wctype("digit"))    // iswdigit(wc)
iswctype(wc, wctype("graph"))    // iswgraph(wc)
iswctype(wc, wctype("lower"))    // iswlower(wc)
iswctype(wc, wctype("print"))    // iswprint(wc)
iswctype(wc, wctype("punct"))    // iswpunct(wc)
iswctype(wc, wctype("space"))    // iswspace(wc)
iswctype(wc, wctype("upper"))    // iswupper(wc)
iswctype(wc, wctype("xdigit"))   // iswxdigit(wc)
```

Returns

4 The **iswctype** function returns nonzero (true) if and only if the value of the wide character `wc` has the property described by `desc`. If `desc` is zero, the **iswctype** function returns zero (false).

Forward references: the **wctype** function (7.30.2.2.2).

7.30.2.2.2 The wctype function

Synopsis

1
```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

2 The `wctype` function constructs a value with type `wctype_t` that describes a class of wide characters identified by the string argument `property`.

3 The strings listed in the description of the `iswctype` function shall be valid in all locales as `property` arguments to the `wctype` function.

Returns

4 If `property` identifies a valid class of wide characters according to the `LC_CTYPE` category of the current locale, the `wctype` function returns a nonzero value that is valid as the second argument to the `iswctype` function; otherwise, it returns zero.

7.30.3 Wide character case mapping utilities

1 The header `<wctype.h>` declares several functions useful for mapping wide characters.

7.30.3.1 Wide character case mapping functions

7.30.3.1.1 The `towlower` function

Synopsis

1
```
#include <wctype.h>
wint_t towlower(wint_t wc);
```

Description

2 The `towlower` function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a wide character for which `iswupper` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswlower` is true, the `towlower` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.30.3.1.2 The `towupper` function

Synopsis

1
```
#include <wctype.h>
wint_t towupper(wint_t wc);
```

Description

2 The `towupper` function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a wide character for which `iswlower` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswupper` is true, the `towupper` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.30.3.2 Extensible wide character case mapping functions

1 The functions `wctrans` and `towctrans` provide extensible wide character mapping as well as case mapping equivalent to that performed by the functions described in the previous subclause (7.30.3.1).

7.30.3.2.1 The `towctrans` function

Synopsis

1
```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

2 The `towctrans` function maps the wide character `wc` using the mapping described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctrans` that returned the value `desc`.

3 Each of the following expressions behaves the same as the call to the wide character case mapping function (7.30.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))    // towlower(wc)
towctrans(wc, wctrans("toupper"))    // towupper(wc)
```

Returns

4 The `towctrans` function returns the mapped value of `wc` using the mapping described by `desc`. If `desc` is zero, the `towctrans` function returns the value of `wc`.

7.30.3.2.2 The `wctrans` function

Synopsis

1
```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Description

2 The `wctrans` function constructs a value with type `wctrans_t` that describes a mapping between wide characters identified by the string argument `property`.

3 The strings listed in the description of the `towctrans` function shall be valid in all locales as `property` arguments to the `wctrans` function.

Returns

4 If `property` identifies a valid mapping of wide characters according to the `LC_CTYPE` category of the current locale, the `wctrans` function returns a nonzero value that is valid as the second argument to the `towctrans` function; otherwise, it returns zero.

7.31 Three-way comparison <stdcompare.h>

0 **NOTE** This clause is formulated with naming schemes that come directly from C++. How these names will be brought into C is still open. On the other hand this specification here may not yet completely be in line with the C++ specification either.

Constraints

1 The operands of three-way comparison shall have function type, function literal type or have a complete object type, that is not a structure type with a flexible array member.

2 If any of the operands has type **`bool`**, so shall the other. If any of the operands has type **`nullptr_t`**, the other operand shall be a null pointer constant, be a pointer, a function designator, a function literal or an array. Otherwise, if any of the operands has pointer type, the other shall have a compatible type, an array, function type or function literal type with a generic type that is compatible, or be a null pointer constant. Otherwise, if any of the operands has function type or function literal type, the other shall have a function type or function literal type that is compatible to the first. Otherwise, if both the operands have array type, their element types shall be compatible. Otherwise, if the operands do not have compatible types they shall have arithmetic type.

Description

3 All three-way comparison that respects the above constraints are well defined and do not raise any floating-point exceptions or signals. If for some border cases a result is explicitly unspecified, this only means that any of the possible values for the type may be chosen for the execution of a specific operation, but that once such a choice has been made and the value has been stored in an object, that value is fixed and will not change by itself.

4 Three-way comparison is specified through a number of result types and values (7.31.1), the operation on primary types (7.31.2), a synthetic operation on aggregate types (7.31.3), and through some supporting macros (7.31.4) and generic functions (7.31.5).

7.31.1 Comparison types

1 The following types shall be defined with the indicated names and values. They are the possible result types of the <=> operator and are pairwise not compatible, but may be the same as or compatible to any standard type as long as that fulfills the necessary requirements. They are either arithmetic types or complete structure types. For each type, the listed constants have different values, and no three-way comparison shall produce a value other than the listed constants.

type	constants	primary type categories
`std::weak_equality`	`std::weak_equality::equivalent` `std::weak_equality::nonequivalent`	opaque types, union types, complex types
`std::strong_equality`	`std::strong_equality::equal` `std::strong_equality::unequal`	function designators, pointers or literals

`std::partial_ordering`	`std::partial_ordering::less`
	`std::partial_ordering::equivalent`
	`std::partial_ordering::greater`
	`std::partial_ordering::unordered`
		real floating types

`std::weak_ordering`	`std::weak_ordering::less`
	`std::weak_ordering::equivalent`
	`std::weak_ordering::greater`	

`std::strong_ordering`	`std::strong_ordering::less`
	`std::strong_ordering::equal`
	`std::strong_ordering::greater`
		integer types, object pointers, `nullptr_t`

The indicated constants all form constant expressions of the indicated type. The constants do not form modifiable lvalues and shall not be used as the operand of a unary & operator.

2 The first two types are collectively called *equality types*, the other three are collectively called *ordering types*, all five are the *three-way comparision types*. The types **`std::weak_equality`**, **`std::weak_ordering`**, **`std::partial_ordering`** are called the *weak comparison types*, **`std::`**

`strong_equality` and `std::strong_ordering` are *strong comparison types*. If the result is a strong comparison type, a result value when comparing two objects that is `std::strong_ordering::equal` or `std::strong_equality::equal` indicates that the two operands compare equal and that their representation bytes when compared individually or by means of `memcmp` will not distinguish the two operands.

3 The comparison types model the possible behavior of the three-way comparison operator when comparing the primary type categories that are listed in the table above. When applied to two operands of the type categories listed in the last column, the result of the three-way comparison operator has the indicated comparison type unless one of the following holds:

— If more than one representation can be the result of storing a null function pointer, comparing function designators, function pointers or function literals yields `std::weak_equality`.

— If more than one representation can be the result of storing a null object pointer, comparing object pointers yields `std::weak_ordering`.

— If for a given integer type, more than one representation can be the result of storing the same value, comparing values of that integer type yields `std::weak_ordering`.[460)]

The macro `std :: isstrong`, see below, can be used to test if a given type gives rise to strong or weak comparison.

4 The feature test macro `__CORE_WEAK_THREEWAY_COMPARISON__` (6.10.8.2) may be used to test if some type categories have weak or strong comparison; if defined the macro expands to an integer constant expression suitable for an `#if` preprocessiong directive such that

```
__CORE_WEAK_THREEWAY_COMPARISON__ ∩ N
```

is non-zero with *N* according to the following table:

N	Condition
1	Multiple byte representations are used for null function pointers.
2	Multiple byte representations are used for null object pointers.
4	There are integer types with inconsistently represented padding bits.
8	There are structure types with inconsistently represented padding bytes, see 7.31.3.

If it is not defined or has the value `0`, wherever this specifications leaves an implementation-defined choice between a strong and a weak three-way comparison, the strong version is chosen.

5 One of the three-way comparison types *T* may be converted implicitly or explicitly to another of these types *S* according to the following table, or by any possible composition of these.

value	conversion	rev.
`std::strong_equality::equal`	`std::weak_equality::equivalent`	yes
`std::strong_equality::unequal`	`std::weak_equality::nonequivalent`	yes
`std::strong_ordering::equal`	`std::strong_equality::equal`	no
`std::strong_ordering::less`	`std::strong_equality::unequal`	no
`std::strong_ordering::greater`	`std::strong_equality::unequal`	no
`std::strong_ordering::less`	`std::weak_ordering::less`	yes
`std::strong_ordering::equal`	`std::weak_ordering::equivalent`	yes
`std::strong_ordering::greater`	`std::weak_ordering::greater`	yes
`std::weak_ordering::less`	`std::partial_ordering::less`	yes
`std::weak_ordering::equivalent`	`std::partial_ordering::equivalent`	yes
`std::weak_ordering::greater`	`std::partial_ordering::greater`	yes

460) This means that types that have no padding bits (such as character types) and the type `bool` will necessarily compare strongly. For types that have padding bits, a strong result type indicates that the implementation guarantees that equal values will always be represented the same.

`std::partial_ordering::equivalent`	`std::weak_equality::equivalent`	no
`std::partial_ordering::less`	`std::weak_equality::nonequivalent`	no
`std::partial_ordering::greater`	`std::weak_equality::nonequivalent`	no
`std::partial_ordering::unordered`	`std::weak_equality::nonequivalent`	no

6 If there is an implicit conversion from type T to type S, T is said to be *stronger* than S, and the *combined type* of the two is S; if they are different types and neither is stronger than the other the combined type is **`std::weak_equality`**. If two values of different conversion type are used as the second and third operand of a conditional expression, the result has the combined type of the two types and the result value is converted to that type.

7 There are also explicit conversions (casts) from weaker types to stronger types when the column "rev." of the table above reads "yes". If the value **`std::partial_ordering::unordered`** is converted to the type **`std::weak_ordering`** the result is never **`std::weak_ordering::equivalent`**, but it is otherwise unspecified which of the other two values is chosen. An explicit conversion from **`std::partial_ordering`** to **`std::strong_ordering`**, is performed as a composition of a conversion to **`std::weak_ordering`** and then to **`std::strong_ordering`**. An explicit conversion from a weak ordering type to **`std::strong_equality`**, is performed as a composition of a conversion to **`std::weak_equality`** and then to **`std::strong_equality`**.

8 Any value of such a type T may be converted to any real type A. For signed integer types and real floating point types, T**`::equal`** (respectively T**`::equivalent`**) converts to the value 0. Any other value converts to a non-zero value that, for floating types, is normal unless specified otherwise; T**`::less`** results in a value that is strictly negative, T**`::greater`**, T**`::unequal`** or T**`::nonequivalent`**, respectively, results in a value that is strictly positive. Which particular value is chosen for such a conversion is unspecified; the result value is fixed once it is stored, but different conversions of the same comparison value, even during the same program execution, may yield different results.[461] If the value is **`std::partial_ordering::unordered`**,[462] the result of such a conversion is the same as (A)**`NAN`** if the implentation has quiet NaNs; otherwise the result is the same as (A)**`INFINITY`**, but no floating-point exception or signal shall be raised. If A is **`bool`**, T**`::equal`** (or T**`::equivalent`**) converts to **`false`** and all other values convert to **`true`**. If A is another unsigned type, the result is the same as to convert to the corresponding signed type, first.

9 Such a conversion to a real type need not to be explicit if the target type is **`bool`**, or if a comparison type is itself used as the operand of an equality operator or of a relational operator where the other operand is an arithmetic constant expression of value 0. Otherwise, whether or not such a conversion is performed implicitly or if it needs a cast operator is implementation-defined.

10 **NOTE** A minimal implementation of these comparison types can be as follows. Here the specific values are chosen such that the usual implicit conversion between arithmetic types lead to the correct results.

```
typedef bool          std::weak_equality;
typedef uintwidth(1)  std::strong_equality;
typedef float         std::partial_ordering;
typedef signed int    std::weak_ordering;
typedef signed char   std::strong_ordering;

[[core::noalias]] inline  bool          std::weak_equality::equivalent    = false;
[[core::noalias]] inline  bool          std::weak_equality::nonequivalent = true;

[[core::noalias]] inline  uintwidth(1)  std::strong_equality::equal       = 0;
[[core::noalias]] inline  uintwidth(1)  std::strong_equality::unequal     = 1;

[[core::noalias]] inline  float         std::partial_ordering::less        = -1.0F;
```

[461]For example, the result may depend on the types and values of the operands of the three-way comparison from which the comparison value originates, but also on the context of the conversion. To encourage optimizations, the intent is to provide as much latitude to the implementation as possible and not to constrain them more as specified, here.

[462]For the types that are listed in the table, such a situation may only occur for floating comparison if one or both operands are a NaN, or for object pointer comparison where the operands have different provenance, including if one of the operands is a null pointer.

```
 [[ core::noalias ]] inline  float       std::partial_ordering::equivalent  =  0.0F;
 [[ core::noalias ]] inline  float       std::partial_ordering::greater     = +1.0F;
#ifdef NAN
 [[ core::noalias ]] inline  float       std::partial_ordering::unordered   = NAN;
#else
 [[ core::noalias ]] inline  float       std::partial_ordering::unordered   = INFINITY;
#endif

 [[ core::noalias ]] inline  signed int  std::weak_ordering::less           = -1;
 [[ core::noalias ]] inline  signed int  std::weak_ordering::equivalent     =  0;
 [[ core::noalias ]] inline  signed int  std::weak_ordering::greater        = +1;

 [[ core::noalias ]] inline  signed char std::strong_ordering::less         = -1;
 [[ core::noalias ]] inline  signed char std::strong_ordering::equal        =  0;
 [[ core::noalias ]] inline  signed char std::strong_ordering::greater      = +1;
```

7.31.2 Primary three-way comparison

1 The result of a three-way operator for the indicated primary type categories is computed as follows. Any operand that is a function designator or literal is converted to the corresponding function pointer. If one operand is an array and the other is not, array-to-pointer conversion is applied to the array. Then, if one of the operands has scalar type a common type is determined as for a conditional operator (6.5.16) and both operands are converted to that type. Now, if the operands of a three-way comparison have the same type *A*, and *A* has one of the indicated type categories, the following rules apply:

— If *A* is an opaque or union type, the result is `std::weak_equality::nonequivalent`.

— If *A* is a function pointer type, the three-way comparison is `std::strong_equality::equal` if the referred functions are the same or if both operands are null pointers, and `std::strong_equality::unequal`, otherwise.

— If *A* is a object pointer type and if both operands are null, the result is `std::strong_ordering::equal`. If one is null and the other is a valid pointer, the result is `std::strong_ordering::less` or `std::strong_ordering::greater` if the null pointer is the first or second operand, respectively. If both pointers are valid, the result is `std::strong_ordering::less`, `std::strong_ordering::equivalent`, or `std::strong_ordering::greater` if the abstract address correponding to the left operand is less than, equal to or greater than the abstract address correponding to the right operand, respectively. Otherwise, the result is unspecified.

— If *A* is `nullptr_t`, the result is `std::strong_ordering::equal`.

— If *A* is `bool`, the result is `std::strong_ordering::equal` if both operands are the same, it is `std::strong_ordering::less` if the left operand is `false` and the right is `true`, `std::strong_ordering::greater` if the values are inverted.

— If *A* is another integer type, the result is `std::strong_ordering::less`, `std::strong_ordering::equal`, or `std::strong_ordering::greater` if the left operand is less than, equal to or greater than the right operand.

— If *A* is a real floating type, the result is `std::partial_ordering::unordered`, `std::partial_ordering::less`, or `std::partial_ordering::greater` if the `isunordered`, `isless` or `isgreater` macros return `true` when called with the operands as arguments. Otherwise, it is `std::partial_ordering::equivalent`. No floating-point exception is raised.

— If *A* is a complex type, the result is determined as for real floating point arrays with two elements, see below, and the result of that three-way comparison is then converted to `std::weak_equality`.

2 If A is an pointer type or an integer type, according to the above the result can be a strong or a weak comparison type. If it is the latter, the result is the specified value after conversion to that weak comparison type.

3 **NOTE** The fact that the three-way comparison type is strong does not indicate the absence of padding bits or bytes. It only indicates that all operations that operate directly on the type will always maintain such padding consistent. Representations that are distinct but that compare equal after lvalue conversion may still exist, but they will never be obtained from operations for the type.

7.31.3 Synthetic three-way comparison

1 If both operands of a three-way comparison have aggregate types, the result is defined by recursively combining the results of comparing corresponding elements (for array types) or members (for structure types).

2 If both operands of a three-way comparison operator have an array type, they have compatible base types, and the result type is the same comparison type as for that base type. Let n_0 and n_1 be the sizes of the left and right operand, respectively. The result value is the value as designated in the following and converted to the so-determined target type.

— If the result type is an equality comparison type: if $n_0 \neq n_1$, the result is `std::strong_equality::unequal`, as it is if $n_0 \equiv n_1$ and if any of the element pairs with same position in their arrays compares unequal or nonequivalent. Otherwise it is `std::strong_equality::equal`.

— If the result type is an ordering comparison type: the elements of the initial parts up to the minimum of n_0 and n_1 are pairwise compared in order. If all of these are equivalent, the result is the result of the three-way comparison n_0 **<=>** n_1. Otherwise the result is the result of the first element-wise comparison that is not equivalent.

To determine the outcome, no array element is accessed that has a position that is beyond the minimum of n_0 and n_1 or beyond the position where the first discrepancy is detected.

3 Otherwise both operands have compatible structure types. The result type is the iterated combined type of all the result types of the member comparisons, unless that type is a strong comparison type, the structure type has padding bytes and there are operations on the structure type that would violate the requirements for strong comparisons. Then result type is the corresponding weak comparison type. The result value is the value as designated in the following and then converted to the so-determined target type.

— If the result type is one of the equality types, the the result value is `std::strong_equality::equal` if all pairwise member comparisions are equal or equivalent, and `std::strong_equality::unequal` otherwise.

— If the result type is one of the ordering types, the result value is `std::strong_ordering::equal` if all member comparisons are equal or equivalent. Otherwise it is the result of the first member conversion (in declaration order) that is not equal or equivalent.

4 **NOTE 1** If the base type of the arrays is a basic type, array comparison is similar to invoking the following generic lambda:

```
[](size_t n₀, const auto A₀[n₀], size_t n₁, const auto A₁[n₁]) {
  decltype(A₀[0] <=> A₁[0]) ret;
  for (size_t i = 0, n = min(n₀, n₁); i < n; ++i) {
    ret = (A₀[i] <=> A₁[i]);
    if (¬ret) return ret;
  }
  ret = (std::strong_ordering)(n₀ <=> n₁);
  return ret;
}
```

Here, the conversion to `std::strong_ordering` is needed if three-way comparison of the type `size_t` is `std::weak_ordering`, namely when `size_t` has padding bits and the implementation doesn't warrant that it handles these padding bits consistently.

5 **NOTE 2** For a structure type without padding, the three-way comparison would iterate over the pairs of members in declaration order:

```
[](const auto A, const auto B) {
  auto ret0 = (A.member0 <=> B.member0);
  auto ret1 = (ret0 ? ret0 : (A.member1 <=> B.member1));
  auto ret2 = (ret1 ? ret1 : (A.member2 <=> B.member2));
  ...
  auto ret = ...;
  return ret;
}
```

Here the **`auto`** definitions of the series of return types ensures that the combined comparison type of all the member comparisons is the final return type for `ret`. The conditional expressions ensure that once a value is detected that is not equal or equivalent, no three-way comparison of other members is evaluated, but that the result value is still converted to the type needed for the return.

6 **EXAMPLE 1** Whether or not two pointers refer to disjoint arrays can be tested with the following function.

```
bool disjoint(size_t n0, void p0[n0], size_t n1, void p1[n1]) {
  std::strong_ordering r0 = (p0 <=> p1+n1);
  std::strong_ordering r1 = (p1 <=> p0+n0);
  return r0 <=> r1;
}
```

If the two pointer arguments are valid, the two three-way comparisons inside the parenthesis are comparisons between valid pointers and thus result in two values of type **`std::strong_ordering`**. If the array sizes are non-zero, and one of the comparisons is **`std::strong_ordering::equal`**, one array follows the other immediately in the address space, and thus the other comparison is different from **`std::strong_ordering::equal`**.

7 If both values are **`std::strong_ordering::less`**, p_0 lays inside the array indicated by p_1; if they are both **`std::strong_ordering::greater`**, p_1 lays inside the one for p_0. Thus, if r_0 and r_1 are equal, the arrays intersect.

8 If r_0 is **`std::strong_ordering::less`** or **`std::strong_ordering::equal`**, and r_1 is different, the array of p_0 lays entirely to the left of the one of p_1; If r_0 is **`std::strong_ordering::equal`** or **`std::strong_ordering::greater`**, and r_1 is different, the array of p_0 lays entirely to the right of the one of p_1. Thus, if r_0 and r_1 are different, the arrays are disjoint.

9 The type **`std::strong_ordering`** is an arithmetic or structure type, and therefore it is a valid operand type to three-way comparison itself. The result of that third three-way comparison can be converted to **`bool`** to know if the first two comparisons had equivalent results or not, and thus if the two arrays intersect or not.

10 **EXAMPLE 2** Three-way comparison can be used for arrays for which only a pointer to the first element and some external size information has been provided. The following demonstrates a functioning implementation of the **`strncmp`** type-generic macro.

```
#define strncmp [](auto s1[], decltype(s1) s2) {                       \
      size_t m = strchr(s1, 0)-s1;               /* the length of s1       */ \
      typedef generic_type(*s1) CT;              /* a (wide) character type */ \
      typedef CT AT[m];                          /* a VLA of character type */ \
      return (int)(                                                      \
           [](AT* S1, AT* S2) [[core::reinterpret]] {  /* pointers to VLA  */ \
                return *S1 <=> *S2;                                      \
           }((AT*)s1, (AT*)s2));                                         \
}
```

If it is called with strings or wide strings, that is with (wide) character arrays that have a null character somewhere, the three-way comparison inside the lambda returns the correct value (of type **`std::strong_ordering`**) because the **`core::reinterpret`** attribute forces an interpretation of the pointers as the addresses of VLAs. This three-way comparison stops at the first character for which the strings differ, and so s_2 will not be accessed outside the bounds, either.

11 The innermost lambda returns type **`std::strong_ordering`**, and that return value is then returned by the outermost lambda. The cast to **`int`** then ensures that the comparison value is correctly converted into an integer. The definition of these types of conversions is sufficiently loose, such that an implementation may rewrite the effective computation of the character comparison followed by a conversion to **`int`** to a difference between character values, as long that fits into **`int`**.

7.31.4 Classification of comparison types and values

7.31.4.1 The `isstrong` macro

Synopsis

1

```
#include <stdcompare.h>
bool std::isstrong(R x);
```

Constraints

2 The argument of the **isstrong** macro shall have a type that is admissible for the three-way comparison operator.

Description

3 The **isstrong** macro determines if the type of the argument yields a strong comparison type when used with the three-way comparison operator. The argument is not evaluated and the result is an integer constant expression.

Returns

4 The **isstrong** macro returns **true** if for the type of its argument no two values with different representations are considered equivalent, and **false** otherwise.

5 **NOTE** This macro may be implemented as

```
#define std::isstrong(X)              \
      generic_selection(((X) <=> (X)), \
            std::strong_ordering: true, \
            std::strong_equality: true, \
            default: false)
```

7.31.4.2 The issymmetric macro

Synopsis

1
```
#include <stdcompare.h>
bool std::issymmetric(R x);
```

Constraints

2 The argument of the **issymmetric** macro shall have a type that is admissible for the three-way comparison operator.

Description

3 The **issymmetric** macro determines if the type of the argument yields an equality comparison type when used with the three-way comparison operator. The argument is not evaluated and the result is an integer constant expression.

Returns

4 The **issymmetric** macro returns **true** if for any values `x` and `y` the the three-way comparisons `x<=>y` and `y<=>x` are interchangeable, and **false** otherwise.

5 **NOTE** This macro may be implemented as

```
#define std::issymmetric(X)           \
      generic_selection(((X) <=> (X)), \
            std::weak_equality: true,   \
            std::strong_equality: true, \
            default: false)
```

7.31.4.3 The toswitch macro

Synopsis

1
```
#include <stdcompare.h>
S std::toswitch(T x);
```

Constraints

2 The argument of the **toswitch** macro shall have a comparison type.

Description

3 The **toswitch** macro converts any comparison value to specific integer values such that the result may be used as a controlling expression of a **switch** statement. For equality types types, S is **uintwidth**(1); for ordering types it is **intwidth**(2). The result is the same value as the expression

```
generic_selection(x,
      std::weak_equality:   (uintwidth(1))(bool)x,
      std::strong_equality: (uintwidth(1))(bool)x,
      default: (intwidth(2))(isfinite((float)x) ? (x < 0 ? -1 : (x ≠ 0)) : -2))
```

only that x is evaluated only once and that the <math.h> and <stdint.h> headers need not to be included. If the argument is a constant expression, the result is an integer constant expression.

Returns

4 The **toswitch** macro returns the values -2, -1, 0, or 1 according to the conversion rules for comparison types.

5 **NOTE** For ordering types, the return type **intwidth**(2) instructs the translator that the admissible return values may be encoded with two value bits, and thus that there is no other possible return value to be expected than the four listed above.

6 **EXAMPLE** The **toswitch** macro can be used effectively without explicitly using the **isfinite** macro or the **intwidth**(2) and the **uintwidth**(1) types and thus without including the <math.h> and <stdint.h> headers. For an ordering type we can use this as follows:

```
      switch (std::toswitch(a <=> b)) {
            case -2: puts("a and b are unordered");          break;
            case -1: puts("a is strictly less than b");      break;
            case  0: puts("a and b are equal/equivalent");   break;
            case +1: puts("a is strictly greater than b");   break;
            default: puts("this is not possible");           break; // diagnostic?
      }
```

For such a **switch** statement with cases for all the four values, the **default** case can never trigger. Thus the translator may omit that case and emit a diagnostic.

7 For an equality type the **toswitch** macro does not bring much more than a conversion to **bool**, but the result is interpreted as an unsigned value with only one bit of information:

```
      switch (std::toswitch(a <=> b)) {
            case 0U: puts("a and b are equal/equivalent");      break;
            case 1U: puts("a and b are unequal/nonequivalent"); break;
            default: puts("this is not possible");              break; // diagnostic?
      }
```

7.31.4.4 The tostrong macro

Synopsis

1
```
      #include <stdcompare.h>
      S std::tostrong(W x);
```

Constraints

2 The argument of the **tostrong** macro shall have a comparison type.

Description

3 The **tostrong** macro converts values of the weak comparison types to the strong types. If the argument type is an equality type the result type is **std**::**strong_equality**, otherwise it is **std**::**strong_ordering**. Values are converted as if by explicit conversion. If the argument is a constant expression, so is the result.

Returns

4 The **tostrong** macro returns the value of type **std::strong_equality** or **std::strong_ordering**, respectively, that corresponds to the comparison semantic of its argument.

5 **NOTE** The **tostrong** macro behaves similar to the follwing:

```
[](auto x) {
      std::strong_ordering ret;
      switch(std::toswitch(x)) {
            default: ret = std::strong_ordering::less;    break;
            case  0: ret = std::strong_ordering::equal;   break;
            case +1: ret = std::strong_ordering::greater; break;
      }
      return generic_selection(x,
            std::strong_ordering:  ret,
            std::weak_ordering:    ret,
            std::partial_ordering: ret,
            std::strong_equality:  (std::strong_equality)ret,
            std::weak_equality:    (std::strong_equality)ret);
      }
```

7.31.5 Searching and sorting utilities

Constraints

1 The type R as parameter of the prototypes for the macros in this clause shall be a type that is admissible for the three-way comparison operator, and such that the result of such an operator is an ordering type.

Description

2 The macros in this clause provide type-safe interfaces analogous to the searching and sorting utilities of 7.22.6 that are based on the three-way comparison operator.

3 The following type is declared

```
#include <stdcompare.h>
typedef int std::compare_t(const void*, const void*);
```

7.31.5.1 The tocompare macro

Synopsis

1
```
#include <stdcompare.h>
Tλ std::tocompare(R x);
```

Description

2 The **tocompare** macro returns a function literal λ that can be converted to a function pointer FP with type **std::compare_t***. It implements a comparison function that is based on the three-way comparison operator for R and such that FP is suitable to be used with the **bsearch** and **qsort** library features. The argument x is not evaluated but only inspected for its type. If FP is called with arguments that point to objects that have a type that is not compatible with the type of x, the behavior is undefined. The result of such a call to FP is strictly negative or strictly postive if the three-way comparison of the two objects returns **less** or **greater**, respectively; it is zero, if the comparison returns **equal** or **equivalent**; it is unspecified, otherwise.

Returns

3 The **tocompare** macro returns a function literal that implements the indicated functionality.

4 **NOTE** A possible implementation of the **tocompare** macro is as follows.

```
#define std::tocompare(X)                                    \
      [](const void* A, const void* B) {                     \
```

```
        const decltype(X)* a = A;                    \
        const decltype(X)* b = B;                    \
        int cmp = (int)(*a <=> *b);                  \
        return cmp;                                  \
    }
```

7.31.5.2 The search type-generic macro

Synopsis

1
```
    #include <stdcompare.h>
    R* std::search(size_t n, R base[n], const R key[1]);
```

Description

2 The **search** type-generic macro provides a type-safe binary search feature with properties similar to **bsearch**. The return value may be converted to a function pointer that points to a function that is compatible to the prototype as indicated in the synopsis.

3 The **search** type-generic macro behaves as if it were defined as follows.

```
#define search                                                              \
[](size_t n, auto base[n],                                                  \
    const generic_type(base[0]) key[1])  [[ core::alias(base) ]] {          \
        auto b = tovoidptr(base+0);                                         \
        auto ret = bsearch(key, b, n, sizeof(base[0]), std::tocompare(base[0])); \
        return (decltype(base+0))ret;                                       \
    }
```

That is, it has the same ordering requirements for `base` as **bsearch** were the ordering is the ordering obtained by the three-way comparison operator on the type *R*. The returned pointer has the same base type (including qualifications) as `base`.

Returns

4 If invoked without arguments, the **search** type-generic macro returns a generic function literal that implements the indicated functionality. If invoked with arguments, the function call returns a pointer to an element of the array that compares equal (or equivalent) to `key[0]`, or a null pointer if no match is found. If two elements compare equal (or equivalent), which element is matched is unspecified.

7.31.5.3 The sort type-generic macro

Synopsis

1
```
    #include <stdcompare.h>
    void std::sort(size_t n, R base[n]);
```

Description

2 The **sort** type-generic macro provides a type safe sort feature with properties similar to **qsort**. The return value may be converted to a function pointer that points to a function that is compatible to the prototype as indicated in the synopsis.

3 The **sort** type-generic macro behaves as if it were defined as follows.

```
#define sort                                                   \
    [](size_t n, auto base[n]) {                               \
        qsort(base, n, sizeof(base[0]), std::tocompare(base[0])); \
    }
```

That is, it has the same ordering requirements for *R* with three-way comparison operator as **qsort**. If the three-way comparison for *R* is **std::partial_ordering** and any of the element pairs in the array may compare as **std::partial_ordering::unordered**, the behavior is undefined.[463)]

Returns

4 If invoked without arguments, the **sort** type-generic macro returns a generic function literal that implements the indicated functionality. If invoked with arguments, the **sort** type-generic macro returns no value.

5 **EXAMPLE** To sort complex arrays in a reproducible way, the lexicographic ordering of real and imaginary part may be used by reinterpreting a complex value as a two-element real vector.

```
#define sort_repro                                                            \
     [](size_t n, auto base[n]) {                                             \
          typedef real_type(base[0]) RT;                    /* the real type */\
          static inline const size_t vsize = sizeof(base[0])/sizeof(RT);      \
          typedef RT VT[vsize];                             /* vector type   */\
          [n](auto b) [[core::reinterpret]] { sort(n, b); }((VT*)base);       \
     }
```

As given, this can be used for all arithmetic types.

463)In particular, if there are such unordered pairs, a resolution could be inconsistent and create cycles in the "ordering" relation. The sort function could then enter an infinite loop. Therefore special care has to be taken for floating-point types, such that the array does not contain NaNs.

7.32 Future library directions

1 The following C library headers are obsolescent and provide no useful feature:

<iso646.h> <stdalign.h> <stdbool.h> <stddef.h> <tgmath.h>

2 The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

~~The function names cerf~~
~~cerfc~~
~~cexp2~~
~~cexpm1~~
~~clog10~~
~~clog1p~~
~~clog2~~
~~clgamma~~
~~ctgamma and the same names suffixed with f or l may be added to the declarations in the header.~~

7.32.1 Character handling <ctype.h>

1 Function names that begin with either **is** or **to**, and a lowercase letter may be added to the declarations in the <ctype.h> header.

7.32.2 Errors <errno.h>

1 Macros that begin with **E** and a digit or **E** and an uppercase letter may be added to the macros defined in the <errno.h> header.

7.32.3 Floating-point environment <fenv.h>

1 Macros that begin with **FE_** and an uppercase letter may be added to the macros defined in the <fenv.h> header.

7.32.4 Format conversion of integer types <inttypes.h>

1 Macros that begin with either **PRI** or **SCN**, and either a lowercase letter or X may be added to the macros defined in the <inttypes.h> header.

2 The function **imaxabs** is an obsolent features. The identifier **imaxdiv** is reserved for external linkage. This restriction may be remove in future versions of this document.

7.32.5 Localization <locale.h>

1 Macros that begin with **LC_** and an uppercase letter may be added to the macros defined in the <locale.h> header.

7.32.6 Mathematics <math.h>

1 Macros that begin with **FP_** or MATH_ and an uppercase letter may be added to the macros defined in the <math.h> header.

2 Use of the **DECIMAL_DIG** macro is an obsolescent feature. A similar type-specific macro, such as **LDBL_DECIMAL_DIG** can be used instead.

3 Function names that begin with **is** and a lowercase letter may be added to the declarations in the <math.h> header.

7.32.7 Signal handling <signal.h>

1 Macros that begin with either **SIG** and an uppercase letter or **SIG_** and an uppercase letter may be added to the macros defined in the <signal.h> header.

7.32.8 Atomics <stdatomic.h>

1 Macros that begin with **ATOMIC_** and an uppercase letter may be added to the macros defined in the <stdatomic.h> header. Typedef names that begin with either **atomic_** or **memory_**, and

a lowercase letter may be added to the declarations in the <stdatomic.h> header. Enumeration constants that begin with **memory_order_** and a lowercase letter may be added to the definition of the **memory_order** type in the <stdatomic.h> header. Function names that begin with **atomic_** and a lowercase letter may be added to the declarations in the <stdatomic.h> header.

2 The ~~macro **ATOMIC_VAR_INIT** is an obsolescent feature.~~

~~The ability to undefine and perhaps then redefine the macros bool, **true**, and **false**~~ possibility that an atomic type name of an atomic integer type defines a different type than the corresponding direct type is an obsolescent feature.

7.32.9 Integer types <stdint.h>

1 Typedef names beginning with **int** or **uint** and ending with **_t** may be added to the types defined in the <stdint.h> header. Macro names beginning with **INT** or **UINT** and ending with **_MAX**, **_MIN**, or **_C** may be added to the macros defined in the <stdint.h> header.

7.32.10 Input/output <stdio.h>

1 Lowercase letters may be added to the conversion specifiers and length modifiers in **fprintf** and **fscanf**. Other characters may be used in extensions.

2 The use of **ungetc** on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.32.11 General utilities <stdlib.h>

1 Function names that begin with **str** and a lowercase letter may be added to the declarations in the <stdlib.h> header.

2 Invoking **realloc** with a size argument equal to zero is an obsolescent feature.

7.32.12 String and storage handling <string.h>

1 Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter may be added to the declarations in the <string.h> header.

7.32.13 Date and time <time.h>

Macros beginning with **TIME_** and an uppercase letter may be added to the macros in the <time.h> header.

7.32.14 Threads <threads.h>

1 Function names, type names, and enumeration constants that begin with either **cnd_**, **mtx_**, **thrd_**, or **tss_**, and a lowercase letter may be added to the declarations in the <threads.h> header.

7.32.15 Extended multibyte and wide character utilities <wchar.h>

1 Function names that begin with **wcs** and a lowercase letter may be added to the declarations in the <wchar.h> header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.

7.32.16 Wide character classification and mapping utilities <wctype.h>

1 Function names that begin with **is** or **to** and a lowercase letter may be added to the declarations in the <wctype.h> header.

Annex A
(informative)
Language syntax summary

1 **NOTE** The notation is described in 6.1.

A.1 Lexical grammar
A.1.1 Lexical elements

(6.4) *token:*
keyword
identifier
constant
string-literal
punctuator

(6.4) *preprocessing-token:*
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) *keyword:* one of

`alignas`	`default`	~~`register`~~ `not`	`union`
`alignof`	`do`	~~`restrict`~~ `not_eq`	`unsigned`
`and`	`double`	`nullptr`	`void`
`and_eq`	`else`	`or`	`volatile`
`auto`	`enum`	`or_eq`	`while`
`bitand`	`extern`	`return`	~~`_Alignas`~~
`bitor`	`false`	`short`	~~`_Alignof`~~
`bool`	`float`	`signed`	~~`_Atomic`~~
`break`	`for`	`sizeof`	~~`_Bool`~~
`case`	`goto`	`static`	~~`_Complex`~~
`char`	`generic_selection`	`static_assert`	~~`_Generic`~~ `xor`
`compl`		`struct`	~~`_Imaginary`~~
`const`	`if`	`switch`	`xor_eq`
`constexpr`	`inline`	`thread_local`	`_Noreturn`
`continue`	`int`	`true`	~~`_Static_assert`~~
`decltype`	`long`	`typedef`	~~`_Thread_local`~~

A.1.3 Identifiers

(6.4.2.1) *identifier:*
identifier-nondigit
identifier identifier-nondigit
identifier digit

(6.4.2.1) *identifier-nondigit:*
nondigit
universal-character-name
other implementation-defined characters

(6.4.2.1) *nondigit:* one of
`_ a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z`

(6.4.2.1) *digit:* one of
`0 1 2 3 4 5 6 7 8 9`

A.1.4 Universal character names

(6.4.3) *universal-character-name:*
`\u` *hex-quad*
`\U` *hex-quad hex-quad*

(6.4.3) *hex-quad:*
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

A.1.5 Constants

(6.4.4) *constant:*
integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

(6.4.4.1) *integer-constant:*
decimal-constant integer-suffix$_{\text{opt}}$
octal-constant integer-suffix$_{\text{opt}}$
hexadecimal-constant integer-suffix$_{\text{opt}}$

(6.4.4.1) *decimal-constant:*
nonzero-digit
decimal-constant digit

(6.4.4.1) *octal-constant:*
`0`
octal-constant octal-digit

(6.4.4.1) *hexadecimal-constant:*
hexadecimal-prefix hexadecimal-digit
hexadecimal-constant hexadecimal-digit

(6.4.4.1) *hexadecimal-prefix:* one of
`0x 0X`

(6.4.4.1) *nonzero-digit:* one of
`1 2 3 4 5 6 7 8 9`

(6.4.4.1) *octal-digit:* one of
`0 1 2 3 4 5 6 7`

(6.4.4.1) *hexadecimal-digit:* one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

(6.4.4.1) *integer-suffix:*
unsigned-suffix long-suffix$_{\text{opt}}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{\text{opt}}$
long-long-suffix unsigned-suffix$_{\text{opt}}$

(6.4.4.1) *unsigned-suffix:* one of
`u U`

(6.4.4.1) *long-suffix:* one of
`l L`

(6.4.4.1) *long-long-suffix:* one of
`ll LL`

(6.4.4.2) *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

(6.4.4.2) *decimal-floating-constant:*
fractional-constant exponent-part$_{\text{opt}}$ *floating-suffix*$_{\text{opt}}$
digit-sequence exponent-part floating-suffix$_{\text{opt}}$

(6.4.4.2) *hexadecimal-floating-constant:*
hexadecimal-prefix hexadecimal-fractional-constant
binary-exponent-part floating-suffix$_{\text{opt}}$
hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix$_{\text{opt}}$

(6.4.4.2) *fractional-constant:*
digit-sequence$_{\text{opt}}$ **`.`** *digit-sequence*
digit-sequence **`.`**

(6.4.4.2) *exponent-part:*
`e` *sign*$_{\text{opt}}$ *digit-sequence*
`E` *sign*$_{\text{opt}}$ *digit-sequence*

(6.4.4.2) *sign:* one of
`+ -`

(6.4.4.2) *digit-sequence:*
digit
digit-sequence digit

(6.4.4.2) *hexadecimal-fractional-constant:*
hexadecimal-digit-sequence$_{\text{opt}}$ **`.`**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **`.`**

(6.4.4.2) *binary-exponent-part:*
`p` *sign*$_{\text{opt}}$ *digit-sequence*
`P` *sign*$_{\text{opt}}$ *digit-sequence*

(6.4.4.2) *hexadecimal-digit-sequence:*
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

(6.4.4.2) *floating-suffix:*
precision-suffix complex-suffix$_{opt}$
complex-suffix precision-suffix$_{opt}$

(6.4.4.2) *precision-suffix:* one of
f l F L

(6.4.4.2) *complex-suffix:* one of
i I

(6.4.4.3) *enumeration-constant:*
identifier

(6.4.4.4) *character-constant:*
encoding-prefix$_{opt}$ **'** *c-char-sequence* **'**

(6.4.4.4) *encoding-prefix:*
u8
u
U
L

(6.4.4.4) *c-char-sequence:*
c-char
c-char-sequence c-char

(6.4.4.4) *c-char:*
any member of the source character set except
the single-quote `'`, backslash `\`, or new-line character
escape-sequence

(6.4.4.4) *escape-sequence:*
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

(6.4.4.4) *simple-escape-sequence:* one of
`\' \" \? \\`
`\a \b \f \n \r \t \v`

(6.4.4.4) *octal-escape-sequence:*
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

(6.4.4.4) *hexadecimal-escape-sequence:*
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

A.1.5.1 Predefined constants

(6.4.4.5) *predefined-constant:* one of
false nullptr true

A.1.6 String literals

(6.4.5) *string-literal:*
 encoding-prefix$_{opt}$ **"** *s-char-sequence*$_{opt}$ **"**

(6.4.5) *s-char-sequence:*
 s-char
 s-char-sequence s-char

(6.4.5) *s-char:*
 any member of the source character set except
 the double-quote ", backslash \, or new-line character
 escape-sequence

A.1.7 Punctuators

(6.4.6) *punctuator:* one of

[ ] ⟦ ⟧ ( ) { } . ~~->~~ →
++ -- & * + - ~ ~~!~~ ¬
~~/ % << >> < > <= >= == != ^ | && ||~~ × / % ⌫ ⌦
<=> < > ≤ ≥ ≡ ≠ ∩ ^ ∪ ∧ ∨
? : ~~:: ; ...~~ :: ; ...
= ~~*=~~ ×= /= %= += -= ~~<<= >>= &= ^= |=~~ ⌫= ⌦= ∩= ^=
∪=
, # ##
<: :> <% %> -> ! << >> <= >= == != | && || :: ... *= <<= >>= &= |= %: %:%:

A.1.8 Header names

(6.4.7) *header-name:*
 < *h-char-sequence* **>**
 " *q-char-sequence* **"**

(6.4.7) *h-char-sequence:*
 h-char
 h-char-sequence h-char

(6.4.7) *h-char:*
 any member of the source character set except
 the new-line character and >

(6.4.7) *q-char-sequence:*
 q-char
 q-char-sequence q-char

(6.4.7) *q-char:*
 any member of the source character set except
 the new-line character and **"**

A.1.9 Preprocessing numbers

(6.4.8) *pp-number:*

digit
. *digit*
pp-number digit
pp-number identifier-nondigit
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

A.2 Phrase structure grammar

A.2.1 Expressions

(6.5.1) *primary-expression:*

identifier
constant
string-literal
(*expression* **)**
generic-selection

(6.5.1.1) *generic-selection:*

~~**_Generic (** *assignment-expression*~~ **generic_selection (** *controlling-expression* **,** *generic-assoc-list* **)**

(6.5.1.1) *generic-assoc-list:*

generic-association
generic-assoc-list **,** *generic-association*

(6.5.1.1) *generic-association:*

type-name **:** *assignment-expression*
default : *assignment-expression*

(6.5.1.1) *controlling-expression:*

expression

(6.5.2) *postfix-expression:*

primary-expression
array-subscript
function-call
member-access
postfix-addition
compound-literal
lambda-expression

(6.5.2.1) *array-subscript:*

postfix-expression **[** *expression* **]**

(6.5.2.2) *function-call:*

postfix-expression **(** *argument-expression-list*$_{\text{opt}}$ **)**

(6.5.2.2) *argument-expression-list:*

assignment-expression
argument-expression-list **,** *assignment-expression*

(6.5.2.3) *member-access:*

postfix-expression **.** *identifier*
postfix-expression ~~**->** *identifier*~~ → *identifier*

(6.5.2.4) *postfix-addition:*
postfix-expression **++**
postfix-expression **--**

(6.5.2.5) *compound-literal:*
(*type-name* **)** **{** *initializer-list* **}**
(*type-name* **)** **{** *initializer-list* **,** **}**

(6.5.2.6) ~~*argument-expression-list:*~~ *lambda-expression:*
~~*assignment-expression*~~ *capture-clause* *parameter-clause*$_{\text{opt}}$ *attribute-specifier-sequence*$_{\text{opt}}$ *function-body*

(6.5.2.6) *capture-clause:*
~~*argument-expression-list* **,**~~ **[** *capture-list*$_{\text{opt}}$ **]**

(6.5.2.6) *capture-list:*
=
identifier-capture
capture-list **,** *identifier-capture*

(6.5.2.6) *identifier-capture:*
identifier
identifier **=** *assignment-expression*

(6.5.2.6) *parameter-clause:*
(*parameter-type-list*$_{\text{opt}}$ **)**

(6.5.3) *unary-expression:*
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator *cast-expression*
sizeof *unary-expression*
sizeof **(** *type-name* **)**
~~**_Alignof**~~ **alignof** **(** *type-name* **)**

(6.5.3) *unary-operator:* one of
& ***** **+** **-** **~** ~~**!**~~ **¬**

(6.5.4) *cast-expression:*
unary-expression
(*type-name* **)** *cast-expression*

(6.5.5) *multiplicative-expression:*
cast-expression
multiplicative-expression ~~*****~~ **×** *cast-expression*
multiplicative-expression **/** *cast-expression*
multiplicative-expression **%** *cast-expression*

(6.5.6) *additive-expression:*
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

(6.5.7) *shift-expression:*
additive-expression
shift-expression ~~**<<**~~ **⌫** *additive-expression*
shift-expression ~~**>>**~~ **⌦** *additive-expression*

(6.5.8) *compare-expression:*
 shift-expression
 compare-expression **<=>** *shift-expression*

(6.5.9) *relational-expression:*
 ~~*shift-expression*~~ *compare-expression*
 relational-expression **<** ~~*shift-expression*~~ *compare-expression*
 relational-expression **>** ~~*shift-expression*~~ *compare-expression*
 relational-expression ~~**<=** *shift-expression*~~ ≤ *compare-expression*
 relational-expression ~~**>=** *shift-expression*~~ ≥ *compare-expression*

(6.5.10) *equality-expression:*
 relational-expression
 equality-expression ~~**==**~~ ≡ *relational-expression*
 equality-expression ~~**!=**~~ ≢ *relational-expression*

(6.5.11) *AND-expression:*
 equality-expression
 AND-expression ~~**&**~~ ∩ *equality-expression*

(6.5.12) *exclusive-OR-expression:*
 AND-expression
 exclusive-OR-expression **^** *AND-expression*

(6.5.13) *inclusive-OR-expression:*
 exclusive-OR-expression
 inclusive-OR-expression ~~**|**~~ ∪ *exclusive-OR-expression*

(6.5.14) *logical-AND-expression:*
 inclusive-OR-expression
 logical-AND-expression ~~**&&**~~ ∧ *inclusive-OR-expression*

(6.5.15) *logical-OR-expression:*
 logical-AND-expression
 logical-OR-expression ~~**||**~~ ∨ *logical-AND-expression*

(6.5.16) *conditional-expression:*
 logical-OR-expression
 logical-OR-expression **?** *expression* **:** *conditional-expression*

(6.5.17) *assignment-expression:*
 conditional-expression
 unary-expression *assignment-operator* *assignment-expression*

(6.5.17) *assignment-operator:* one of
 = ~~***=**~~ **×=** **/=** **%=** **+=** **-=** ~~**<<=** **>>=** **&=** **^=** **|=**~~ **⌫=** **⌦=** **∩=** **^=** **∪=**

(6.5.18) *expression:*
 assignment-expression
 expression **,** *assignment-expression*

(6.6) *constant-expression:*
 conditional-expression

A.2.2 Declarations

(6.7) *declaration:*
 declaration-specifiers *init-declarator-list*$_{opt}$ **;**
 attribute-specifier-sequence declaration-specifiers *init-declarator-list* **;**
 static_assert-declaration
 attribute-declaration

(6.7) *declaration-specifiers:*
 declaration-specifier *attribute-specifier-sequence*$_{opt}$
 declaration-specifier *declaration-specifiers*

(6.7) *declaration-specifier:*
storage-class-specifier
type-specifier-qualifier
~~*function-specifier*~~ **`constexpr`**
`inline`
`_Noreturn`

(6.7) *init-declarator-list:*
init-declarator
init-declarator-list **`,`** *init-declarator*

(6.7) *init-declarator:*
declarator
declarator **`=`** *initializer*

(6.7) *attribute-declaration:*
attribute-specifier-sequence **`;`**

(6.7.1) *storage-class-specifier:*
`typedef`
`extern`
`static`
~~**`_Thread_local`**~~ **`thread_local`**
`auto`
~~**`register`**~~

(6.7.2) *type-specifier:*
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
~~**`_Bool`**~~
~~**`_Complex`**~~ **`bool`**
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
decltype-specifier
`complex_type`, **`real_type`** or **`generic_type`** specifier macros

(6.7.2.1) *struct-or-union-specifier:*
struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **`{`** *member-declaration-list* **`}`**
struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*

(6.7.2.1) *struct-or-union:*
`struct`
`union`

(6.7.2.1) *member-declaration-list:*
member-declaration
member-declaration-list member-declaration

(6.7.2.1) *member-declaration:*
attribute-specifier-sequence$_{\text{opt}}$ *specifier-qualifier-list member-declarator-list*$_{\text{opt}}$ **`;`**
static_assert-declaration

(6.7.2.1) *specifier-qualifier-list:*
type-specifier-qualifier attribute-specifier-sequence$_{\text{opt}}$
type-specifier-qualifier specifier-qualifier-list

(6.7.2.1) *type-specifier-qualifier:*
type-specifier
type-qualifier
alignment-specifier

(6.7.2.1) *member-declarator-list:*
member-declarator
member-declarator-list **,** *member-declarator*

(6.7.2.1) *member-declarator:*
declarator
bit-field

(6.7.2.1) *bit-field:*
declarator$_{\text{opt}}$ **:** *constant-expression*

(6.7.2.2) *enum-specifier:*
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **, }**
enum *identifier*
(6.7.2.2) *enumerator-list:*
enumerator
enumerator-list **,** *enumerator*
(6.7.2.2) *enumerator:*
enumeration-constant attribute-specifier-sequence$_{\text{opt}}$
enumeration-constant attribute-specifier-sequence$_{\text{opt}}$ **=** *constant-expression*
(6.7.2.4) *atomic-type-specifier:*
~~**_Atomic**~~ **atomic_type** **(** *type-name* **)**
(6.7.3) *type-qualifier:*
const
~~**restrict**~~
~~**volatile**~~
~~**_Atomic** *function-specifier:*~~
~~**inline**~~
~~**_Noreturn**~~
volatile
(6.7.6) *alignment-specifier:*
~~**_Alignas**~~ **alignas** **(** *type-name* **)**
~~**_Alignas**~~ **alignas** **(** *constant-expression* **)**

(6.7.8) *declarator:*
pointer$_{\text{opt}}$ *direct-declarator*

(6.7.8) *direct-declarator:*
identifier attribute-specifier-sequence$_{\text{opt}}$
(*declarator* **)**
array-declarator attribute-specifier-sequence$_{\text{opt}}$
function-declarator attribute-specifier-sequence$_{\text{opt}}$
(6.7.8.1) *pointer:*
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *pointer*

(6.7.8.2) *array-declarator:*
direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[** `static` *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* `static` *assignment-expression* **]**
~~*direct-declarator* [*type-qualifier-list*$_{opt}$ *]~~

(6.7.8.3) *function-declarator:*
direct-declarator **(** *parameter-type-list*$_{opt}$ **)**

~~*pointer:*~~
~~* *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$~~
~~* *attribute-specifier-sequence*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*~~ (6.7.8.3) *type-qualifier-list:*
type-qualifier
type-qualifier-list type-qualifier
(6.7.8.3) *parameter-type-list:*
parameter-list
parameter-list **,** ~~...~~ ...
(6.7.8.3) *parameter-list:*
parameter-declaration
parameter-list **,** *parameter-declaration*
(6.7.8.3) *parameter-declaration:*
attribute-specifier-sequence$_{opt}$ *declaration-specifiers declarator*
attribute-specifier-sequence$_{opt}$ *declaration-specifiers abstract-declarator*$_{opt}$
(6.7.9) *type-name:*
specifier-qualifier-list abstract-declarator$_{opt}$
(6.7.9) *abstract-declarator:*
pointer
pointer$_{opt}$ *direct-abstract-declarator*
(6.7.9) *direct-abstract-declarator:*
(*abstract-declarator* **)**
array-abstract-declarator attribute-specifier-sequence$_{opt}$
function-abstract-declarator attribute-specifier-sequence$_{opt}$
(6.7.9) *array-abstract-declarator:*
direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-abstract-declarator$_{opt}$ **[** `static` *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* `static` *assignment-expression* **]**
direct-abstract-declarator$_{opt}$ **[** * **]**

(6.7.9) *function-abstract-declarator:*
direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

(6.7.10) *typedef-name:*
identifier

(6.7.11) *decltype-specifier:*
`decltype` **(** *identification* **)**
`decltype` **(** *value-expression* **)**

(6.7.11) *identification:*
identifier
member-access

(6.7.11) *value-expression:*
expression

(6.7.12) *initializer:*

assignment-expression
{ *initializer-list*$_{opt}$ **}**
{ *initializer-list* **, }**

(6.7.12) *initializer-list:*

designation$_{opt}$ *initializer*
initializer-list **,** *designation*$_{opt}$ *initializer*

(6.7.12) *designation:*

designator-list **=**

(6.7.12) *designator-list:*

designator
designator-list designator

(6.7.12) *designator:*

[*constant-expression* **]**
. *identifier*

(6.7.14) *static_assert-declaration:*

~~`_Static_assert`~~ `static_assert` **(** *constant-expression* **,** *string-literal* **) ;**
~~`_Static_assert`~~ `static_assert` **(** *constant-expression* **) ;**

(6.7.15.1) *attribute-specifier-sequence:*

attribute-specifier-sequence$_{opt}$ *attribute-specifier*

(6.7.15.1) *attribute-specifier:*

~~**[[** *attribute-list* **]]**~~ [[*attribute-list*]]

(6.7.15.1) *attribute-list:*

attribute$_{opt}$
attribute-list **,** *attribute*$_{opt}$

(6.7.15.1) *attribute:*

attribute-token attribute-argument-clause$_{opt}$

(6.7.15.1) *attribute-token:*

standard-attribute
attribute-prefixed-token

(6.7.15.1) *standard-attribute:*

identifier

(6.7.15.1) *attribute-prefixed-token:*

attribute-prefix ~~**::** *identifier*~~ :: *attribute-suffix*

(6.7.15.1) *attribute-prefix:*

identifier

(6.7.15.1) *attribute-suffix:*

identifier

(6.7.15.1) *core-attribute:*

`core` :: *identifier*

(6.7.15.1) *attribute-argument-clause:*

(*balanced-token-sequence*$_{opt}$ **)**

(6.7.15.1) *balanced-token-sequence:*

balanced-token
balanced-token-sequence balanced-token

(6.7.15.1) *balanced-token:*

(*balanced-token-sequence*$_{opt}$ **)**
[*balanced-token-sequence*$_{opt}$ **]**
{ *balanced-token-sequence*$_{opt}$ **}**
any token other than a parenthesis, a bracket, or a brace

(6.7.15.3) *core-storage-attribute:*
    core **::** *identifier* *attribute-argument-clause*$_{\text{opt}}$
(6.7.15.4) *core-function-attribute:*
    core **::** *identifier* *attribute-argument-clause*$_{\text{opt}}$
(6.7.15.4.1)
    # pragma **CORE FUNCTION_ATTRIBUTE** *attribute*
    # pragma **CORE FUNCTION_ATTRIBUTE** *attribute-token* *on-off-switch*

A.2.3 Statements

(6.8) *statement:*
    labeled-statement
    expression-statement
    attribute-specifier-sequence$_{\text{opt}}$ *compound-statement*
    attribute-specifier-sequence$_{\text{opt}}$ *selection-statement*
    attribute-specifier-sequence$_{\text{opt}}$ *iteration-statement*
    attribute-specifier-sequence$_{\text{opt}}$ *jump-statement*
(6.8.1) *labeled-statement:*
    attribute-specifier-sequence$_{\text{opt}}$ *identifier* **:** *statement*
    attribute-specifier-sequence$_{\text{opt}}$ **case** *constant-expression* **:** *statement*
    attribute-specifier-sequence$_{\text{opt}}$ **default** **:** *statement*
(6.8.2) *compound-statement:*
    { *block-item-list*$_{\text{opt}}$ **}**
(6.8.2) *block-item-list:*
    block-item
    block-item-list *block-item*
(6.8.2) *block-item:*
    declaration
    statement

(6.8.3) *expression-statement:*
    expression$_{\text{opt}}$ **;**
    attribute-specifier-sequence *expression* **;**

(6.8.4) *selection-statement:*
    if **(** ~~*expression* **)** *statement*~~ *init-statement*$_{\text{opt}}$ *controlling-expression* **)** *consequent-body*
    if **(** ~~*expression* **)** *statement* **else** *statement*~~ *init-statement*$_{\text{opt}}$ *controlling-expression* **)** *consequent-body* **else** *alternative-body*
    switch **(** ~~*expression* **)**~~ *controlling-expression* **)** *switch-body*
(6.8.4) *init-statement:*
    expression-statement
    declaration

(6.8.4) *consequent-body:*
    statement

(6.8.4) *alternative-body:*
    statement

(6.8.4) *switch-body:*
    statement

(6.8.5) *iteration-statement:*

while **(** ~~*expression* **)** *statement*~~ *controlling-expression* **)** *loop-body*
do ~~*statement* **while** **(** *expression*~~ *loop-body* **while** **(** *controlling-expression* **)** **;**
for **(** ~~*expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **)** *statement*~~ *init-statement* *controlling-expression*$_{\text{opt}}$ **;** *iteration-expression* **)** *loop-body*

(6.8.5) *loop-body:*

statement

(6.8.5) *iteration-expression:*

~~**for** **(** *declaration* *expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **)** *statement*~~ *expression*$_{\text{opt}}$

(6.8.6) *jump-statement:*

goto *identifier* **;**
continue **;**
break **;**
return *expression*$_{\text{opt}}$ **;**

A.2.4 External definitions

(6.9) *translation-unit:*

external-declaration
translation-unit external-declaration

(6.9) *external-declaration:*

function-definition
declaration

(6.9.1) *function-definition:*

attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers declarator function-body*

(6.9.1) *function-body:*

compound-statement

A.3 Preprocessing directives

(6.10) *preprocessing-file:*

group$_{\text{opt}}$

(6.10) *group:*

group-part
group group-part

(6.10) *group-part:*

if-section
control-line
text-line
*non-directive*

(6.10) *if-section:*

if-group elif-groups$_{\text{opt}}$ *else-group*$_{\text{opt}}$ *endif-line*

(6.10) *if-group:*

 # if ~~*constant-expression*~~ *controlling-expression new-line group*$_{\text{opt}}$
 # ifdef *identifier new-line group*$_{\text{opt}}$
 # ifndef *identifier new-line group*$_{\text{opt}}$

(6.10) *elif-groups:*

elif-group
elif-groups elif-group

(6.10) *elif-group:*

 # elif ~~*constant-expression*~~ *controlling-expression new-line group*$_{\text{opt}}$

(6.10) *else-group:*

  # else *new-line group*$_{opt}$

(6.10) *endif-line:*

  # endif *new-line*

(6.10) *control-line:*

  # include *pp-tokens new-line*
  # define *identifier replacement-list new-line*
  # define *identifier lparen identifier-list*$_{opt}$ **)**
    replacement-list new-line
  # define *identifier lparen* ~~**...**~~ **...** **)** *replacement-list new-line*
  # define *identifier lparen identifier-list* **,** ~~**...**~~ **...** **)**
    replacement-list new-line
  # undef *identifier new-line*
  # line *pp-tokens new-line*
  # error *pp-tokens*$_{opt}$ *new-line*
  # pragma *pp-tokens*$_{opt}$ *new-line*
  # *new-line*

(6.10) *text-line:*

  pp-tokens$_{opt}$ *new-line*

(6.10) *non-directive:*

  pp-tokens new-line

(6.10) *lparen:*

  a **(** character not immediately preceded by white space

(6.10) *replacement-list:*

  pp-tokens$_{opt}$

(6.10) *pp-tokens:*

  preprocessing-token
  pp-tokens preprocessing-token

(6.10) *new-line:*

  the new-line character

(6.10) *identifier-list:*

  identifier
  identifier-list **,** *identifier*

(6.10.6) *on-off-switch:* one of

  ON **OFF** **DEFAULT**

A.4 Floating-point subject sequence

A.4.1 NaN char sequence

(7.22.1.3) *n-char-sequence*:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit ~~*n-wchar-sequence:*~~
  ~~*digit*~~
  ~~*nondigit*~~
  ~~*n-wchar-sequence digit*~~
  ~~*n-wchar-sequence nondigit*~~

Annex B
(informative)
Library summary

B.1 Diagnostics <assert.h>

NDEBUG

```
void assert(scalar expression) [[ core::evaluates(stderr) ]] ;
```

B.2 Complex <complex.h>

__CORE_NO_COMPLEX__
__CORE_VERSION_COMPLEX_H__
__STDC_NO_COMPLEX__

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

B.3 Character handling <ctype.h>

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(locale)
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
constexpr int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
constexpr int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

B.4 Errors <errno.h>

EDOM EILSEQ ERANGE errno

B.5 Floating-point environment <fenv.h>

__CORE_VERSION_FENV_H__
fenv_t
fexcept_t
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
FE_DFL_ENV

```
#pragma CORE FUNCTION_ATTRIBUTE core::idempotent
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(fenv)
#pragma STDC FENV_ACCESS on-off-switch
int feclearexcept(int excepts)
      [[ core::modifies(fenv) ]] ;
int fegetexceptflag(fexcept_t * [[ core::writethrough ]] flagp, int excepts);
```

```
int feraiseexcept(int excepts) [[ core::modifies(fenv) ]] ;
int fesetexceptflag(const fexcept_t *flagp, int excepts)
        [[ core::modifies(fenv) ]] ;
int fetestexcept(int excepts);
int fegetround(void);
int fesetround(int round)
        [[ core::modifies(fenv) ]] ;
int fegetenv(fenv_t * [[ core::writethrough ]] envp);
int feholdexcept(fenv_t *envp)
        [[ core::modifies(fenv) ]] ;
int fesetenv(const fenv_t *envp)
        [[ core::modifies(fenv) ]] ;
int feupdateenv(const fenv_t *envp);
        [[ core::modifies(fenv) ]] ;
```

B.6 Characteristics of floating types <float.h>

FLT_ROUNDS	DBL_DIG	FLT_MAX
FLT_EVAL_METHOD	LDBL_DIG	DBL_MAX
FLT_HAS_SUBNORM	FLT_MIN_EXP	LDBL_MAX
DBL_HAS_SUBNORM	DBL_MIN_EXP	FLT_EPSILON
LDBL_HAS_SUBNORM	LDBL_MIN_EXP	DBL_EPSILON
FLT_RADIX	FLT_MIN_10_EXP	LDBL_EPSILON
FLT_MANT_DIG	DBL_MIN_10_EXP	FLT_MIN
DBL_MANT_DIG	LDBL_MIN_10_EXP	DBL_MIN
LDBL_MANT_DIG	FLT_MAX_EXP	LDBL_MIN
FLT_DECIMAL_DIG	DBL_MAX_EXP	FLT_TRUE_MIN
DBL_DECIMAL_DIG	LDBL_MAX_EXP	DBL_TRUE_MIN
LDBL_DECIMAL_DIG	FLT_MAX_10_EXP	LDBL_TRUE_MIN
DECIMAL_DIG	DBL_MAX_10_EXP	
FLT_DIG	LDBL_MAX_10_EXP	

B.7 Format conversion of integer types <inttypes.h>

__CORE_VERSION_INTTYPES_H__	imaxdiv_t

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR
PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR
SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR
SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

B.8 Characteristics of integer types <limits.h>

BOOL_WIDTH	ULLONG_WIDTH	CHAR_MIN	LONG_MAX
CHAR_BIT	INT_BITFIELD_MAX	CHAR_WIDTH	LONG_MIN
USHRT_WIDTH	MB_LEN_MAX	INT_MAX	LONG_WIDTH
UINT_WIDTH	BOOL_MAX	INT_MIN	LLONG_MAX
ULONG_WIDTH	CHAR_MAX	INT_WIDTH	LLONG_MIN

LLONG_WIDTH SHRT_MAX USHRT_MAX
SCHAR_MAX SHRT_MIN UINT_MAX
SCHAR_MIN UCHAR_MAX ULONG_MAX
SCHAR_WIDTH UCHAR_WIDTH ULLONG_MAX

B.9 Localization <locale.h>

struct lconv LC_COLLATE LC_MONETARY LC_TIME
LC_ALL LC_CTYPE LC_NUMERIC

```
char *setlocale(int category, const char *locale)
      [[ core::unsequenced, core::modifies(locale) ]] ;
struct lconv *localeconv(void) [[ unsequenced, core::evaluates(locale) ]] ;
```

B.10 Mathematics <math.h>

__CORE_VERSION_MATH_H__ NAN FP_FAST_FMAF
float_t FP_INFINITE FP_FAST_FMAL
double_t FP_NAN FP_ILOGB0
HUGE_VAL FP_NORMAL FP_ILOGBNAN
HUGE_VALF FP_SUBNORMAL MATH_ERRNO
HUGE_VALL FP_ZERO MATH_ERREXCEPT
INFINITY FP_FAST_FMA math_errhandling

```
#pragma CORE FUNCTION_ATTRIBUTE core::unsequenced
#pragma CORE FUNCTION_ATTRIBUTE core::modifies(errno, fenv)
#pragma STDC FP_CONTRACT on-off-switch
constexpr int fpclassify(R x);
constexpr bool isfinite(R x);
constexpr bool isinf(R x);
constexpr bool isnan(R x);
constexpr bool isnormal(R x);
constexpr bool signbit(R x);
constexpr F acos(R x);
constexpr F asin(R x);
constexpr F atan(R x);
constexpr F atan2(R x, S y);
constexpr F cos(R x);
constexpr F sin(R x);
constexpr F tan(R x);
constexpr F acosh(R x);
constexpr F asinh(R x);
constexpr F atanh(R x);
constexpr F cosh(R x);
constexpr F sinh(R x);
constexpr F tanh(R x);
constexpr F exp(R x);
constexpr F exp2(R x);
constexpr F expm1(R x);
F frexp(R x, int * [[ core::writethrough ]] exp);
constexpr int ilogb(R x);
constexpr F ldexp(R x, int k);
constexpr F log(R x);
constexpr F log10(R x);
constexpr F log1p(R x);
constexpr F log2(R x);
constexpr F logb(R x);
```

```
R modf(Q value, R * [[ core::writethrough ]] iptr);
constexpr F scalbn(R x, Z n);
constexpr F cbrt(R x);
constexpr R fabs(R x);
constexpr F hypot(R x, S y);
constexpr F pow(R x, S y);
constexpr F sqrt(R x);
constexpr U abs(R x);
constexpr U abs²(R x);
constexpr F erf(R x);
constexpr F erfc(R x);
constexpr F lgamma(R x);
constexpr F tgamma(R x);
constexpr F ceil(R x);
constexpr F floor(R x);
constexpr F nearbyint(R x);
constexpr F rint(R x);
constexpr long int lrint(R x);
constexpr long long int llrint(R x);
constexpr F round(R x);
constexpr long int lround(R x);
constexpr long long int llround(R x);
constexpr F trunc(R x);
constexpr F fmod(R x, S y);
constexpr F remainder(R x, S y);
F remquo(R x, S y, int *quo);
constexpr F copysign(R x, S y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
constexpr F nextafter(R x, F y);
constexpr F nexttoward(R x, long double y);
constexpr F carg(C z);
constexpr C conj(C z);
constexpr C cproj(C z);
constexpr F fdim(R x, S y);
constexpr F fmax(R x, S y);
constexpr F fmin(R x, S y);
constexpr U math_pdiff(R x, S y);
constexpr Q max(R x, S y);
constexpr Q min(R x, S y);
constexpr F fma(R x, S y, T z);
constexpr bool isgreater(R x, S y);
constexpr bool isgreaterequal(R x, S y);
constexpr bool isless(R x, S y);
constexpr bool islessequal(R x, S y);
constexpr bool islessgreater(R x, S y);
constexpr bool isunordered(R x, S y);
constexpr bool isice(A x);      // x is an integer constant expression
constexpr bool isvla(A x);      // x is an variably modified array
constexpr bool iscomplex(A x);  // x has complex type
constexpr bool isconstant(A x); // x is a scalar constant expression
constexpr bool isextended(A x); // x has an extended integer type
constexpr bool isfloating(A x); // x has floating type
constexpr bool isfunction(A x); // x has function or function pointer type
constexpr bool isinteger(A x);  // x has integer type
constexpr bool isnarrow(A x);   // x has a narrow integer type
constexpr bool isnull(A x);     // x is integer and a null pointer constant
constexpr bool ispointer(A x);  // x has pointer, array or function type
constexpr bool isreal(A x);     // x has real type
constexpr bool issigned(A x);   // x has signed type
```

```
constexpr bool isunsigned(A x); // x has unsigned type
constexpr bool iswide(A x);     // x has a wide integer type
constexpr A tohighest(A x);     // maximum of the type of x
constexpr A tolowest(A x);      // minimum of the type of x
constexpr A toone(A x);         // the difference from 1 to the smallest
                                // value strictly greater than 1
constexpr A tozero(A x);        // the difference from 0 to the
                                // smallest value strictly greater than 0
```

B.11 Nonlocal jumps <setjmp.h>

jmp_buf

```
int setjmp( [[ core::writethrough ]] jmp_buf env);
_Noreturn void longjmp(jmp_buf env, int val);
```

B.12 Signal handling <signal.h>

sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM

```
sighandler_t signal(int signum, sighandler_t handler)  [[ core::modifies(errno) ]] ;
int raise(int sig);
```

B.13 Variable arguments <stdarg.h>

va_list

```
type va_arg(va_list ap, type);
void va_copy( [[ core::writethrough ]] va_list dest, va_list src);
void va_end(va_list ap);
void va_start( [[ core::writethrough ]] va_list ap, parmN);
```

B.14 Atomics <stdatomic.h>

__CORE_NO_ATOMICS__
__CORE_VERSION_STDATOMIC_H__

ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
memory_order
atomic_flag
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
atomic_bool
atomic_char
atomic_schar
atomic_uchar
atomic_short
atomic_ushort
atomic_int
atomic_uint
atomic_long
atomic_ulong
atomic_llong
atomic_ullong
atomic_char16_t
atomic_char32_t
atomic_wchar_t
atomic_int_least8_t
atomic_uint_least8_t
atomic_int_least16_t
atomic_uint_least16_t
atomic_int_least32_t
atomic_uint_least32_t
atomic_int_least64_t
atomic_uint_least64_t
atomic_int_fast8_t
atomic_uint_fast8_t
atomic_int_fast16_t
atomic_uint_fast16_t
atomic_int_fast32_t
atomic_uint_fast32_t
atomic_int_fast64_t
atomic_uint_fast64_t
atomic_intptr_t
atomic_uintptr_t
atomic_size_t
atomic_ptrdiff_t
atomic_intmax_t
atomic_uintmax_t

```
void atomic_init(A *obj, C value);
type kill_dependency(type y);
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);
bool atomic_is_lock_free(const A *obj) [[ core::reentrant ]] ;
void atomic_store(A *object, C desired);
void atomic_store_explicit(A *object, C desired, memory_order order);
C atomic_load(const A *object);
C atomic_load_explicit(const A *object, memory_order order);
C atomic_exchange(A *object, C desired);
C atomic_exchange_explicit(A *object, C desired, memory_order order);
bool atomic_compare_exchange_strong(A *object, C *expected, C desired);
bool atomic_compare_exchange_strong_explicit(A *object,
                C *expected, C desired, memory_order success, memory_order failure);
bool atomic_compare_exchange_weak(A *object, C *expected, C desired);
bool atomic_compare_exchange_weak_explicit(A *object,
                C *expected, C desired, memory_order success, memory_order failure);
C atomic_fetch_key(A *object, M operand);
C atomic_fetch_key_explicit(A *object, M operand, memory_order order);
C atomic_key_fetch(A *object, M operand);
C atomic_key_fetch_explicit(A *object, M operand, memory_order order);
bool atomic_flag_test_and_set(A *object);
bool atomic_flag_test_and_set_explicit(A *object, memory_order order);
void atomic_flag_clear(A *object);
void atomic_flag_clear_explicit(A *object, memory_order order);
```

B.15 Integer types <stdint.h>

`__CORE_VERSION_STDINT_H__`	INT*N*_MIN	UINTMAX_MAX
INT_LEAST_WIDTH_MAX	INT*N*_MAX	PTRDIFF_MIN
int*N*_t	UINT*N*_MAX	PTRDIFF_MAX
uint*N*_t	INT_LEAST*N*_MIN	SIG_ATOMIC_MIN
int_least*N*_t	INT_LEAST*N*_MAX	SIG_ATOMIC_MAX
uint_least*N*_t	UINT_LEAST*N*_MAX	SIZE_MAX
int_fast*N*_t	INT_FAST*N*_MIN	WCHAR_MIN
uint_fast*N*_t	INT_FAST*N*_MAX	WCHAR_MAX
intwidth(*M*)	UINT_FAST*N*_MAX	WINT_MIN
uintwidth(*M*)	INTPTR_MIN	WINT_MAX
intptr_t	INTPTR_MAX	INT*N*_C(*value*)
uintptr_t	UINTPTR_MAX	UINT*N*_C(*value*)
intmax_t	INTMAX_MIN	INTMAX_C(*value*)
uintmax_t	INTMAX_MAX	UINTMAX_C(*value*)

B.16 Input/output <stdio.h>

FILE	BUFSIZ	SEEK_CUR	stdin
fpos_t	EOF	SEEK_END	stdout
_IOFBF	FOPEN_MAX	SEEK_SET	
_IOLBF	FILENAME_MAX	TMP_MAX	
_IONBF	L_tmpnam	stderr	

```
int remove(const char *filename) [[ core::modifies(fopen) ]] ;
int rename(const char *old, const char *new) [[ core::modifies(fopen) ]] ;
FILE *tmpfile(void) [[ core::modifies(fopen) ]] ;
char * [[ core::alias ]] tmpnam(char *s) [[ core::alias(s), core::modifies(fopen) ]] ;
int fclose(FILE *stream) [[ core::modifies(fopen) ]] ;
int fflush(FILE *stream) [[ core::modifies(fopen) ]] ;
```

```
FILE *fopen(const char * [[ core::noalias ]]  filename, const char * [[ core::noalias ]]  mode)
      [[ core::modifies(fopen) ]] ;
FILE *freopen(const char * [[ core::noalias ]]  filename, const char * [[ core::noalias ]]  mode,
      FILE * [[ core::noalias ]]  stream)  [[ core::modifies(fopen) ]] ;
void setbuf(FILE * [[ core::noalias ]]  stream, char * [[ core::noalias ]]  buf);
int setvbuf(FILE * [[ core::noalias ]]  stream, char * [[ core::noalias ]]  buf, int mode, size_t
     size);
int printf(const char * [[ core::noalias ]]  format, ...)  [[ core::evaluates(stdout) ]] ;
int scanf(const char * [[ core::noalias ]]  format, ...)  [[ core::evaluates(stdin) ]] ;
int snprintf(char * [[ core::noalias ]]  s, size_t n, const char * [[ core::noalias ]]  format,
    ...);
int sprintf(char * [[ core::noalias, core::writethrough ]]  s, const char * [[ core::noalias ]]
    format, ...);
int sscanf(const char * [[ core::noalias ]]  s, const char * [[ core::noalias ]]  format, ...);
int vfprintf(FILE * [[ core::noalias ]]  stream, const char * [[ core::noalias ]]  format,
    va_list arg)
      [[ core::modifies(errno) ]] ;
int vfscanf(FILE * [[ core::noalias ]]  stream, const char * [[ core::noalias ]]  format, va_list
     arg);
int vprintf(const char * [[ core::noalias ]]  format, va_list arg)
      [[ core::modifies(errno) ]] ;
int vscanf(const char * [[ core::noalias ]]  format, va_list arg);
int vsnprintf(char * [[ core::noalias, core::writethrough ]]  s, size_t n,
      const char * [[ core::noalias ]]  format, va_list arg);
int vsprintf(char * [[ core::noalias, core::writethrough ]]  s,
      const char * [[ core::noalias ]]  format, va_list arg)
            [[ core::modifies(errno) ]] ;
int vsscanf(const char * [[ core::noalias ]]  s, const char * [[ core::noalias ]]  format,
    va_list arg);
int fgetc(FILE *stream);
char *fgets(char * [[ core::noalias, core::writethrough ]]  s, int n, FILE * [[ core::noalias ]]
    stream);
int fputc(int c, FILE *stream);
int fputs(A x, FILE * [[ core::noalias ]]  stream)
 [[ core::stateless ]] ;
int fputs(A x, FILE * [[ core::noalias ]]  stream, A x, UT baseflags)
 [[ core::stateless ]] ;
int fputs(A x, FILE * [[ core::noalias ]]  stream, A x, UT baseflags,
      const char*[[core::noalias ]]  glue)
 [[ core::stateless ]] ;
int getc(FILE *stream);
int getchar(void)  [[ core::evaluates(stdin) ]] ;
int putc(int c, FILE *stream);
int putchar(int c)  [[ core::evaluates(stdout) ]] ;
int puts(A x)
 [[ core::modifies(stdout), core::stateless ]] ;
int puts(A x, UT baseflags)
 [[ core::modifies(stdout), core::stateless ]] ;
int puts(A x, UT baseflags, const char* [[ core::noalias ]] glue)
 [[ core::modifies(stdout), core::stateless ]] ;
int ungetc(int c, FILE *stream);
size_t fread(void * [[ core::noalias, core::writethrough ]]  ptr, size_t size, size_t nmemb,
      FILE * [[ core::noalias ]]  stream);
size_t fwrite(const void * [[ core::noalias ]]  ptr, size_t size, size_t nmemb,
      FILE * [[ core::noalias ]]  stream);
int fgetpos(FILE * [[ core::noalias ]]  stream, fpos_t * [[ core::noalias ]]  pos)
      [[ core::modifies(errno) ]] ;
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos)
      [[ core::modifies(errno) ]] ;
long int ftell(FILE *stream)  [[ core::modifies(errno) ]] ;
```

```
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s) [[core::evaluates(errno, stderr)]];
int fprintf(FILE * [[core::noalias]] stream, const char * [[core::noalias]] format, ...)
      [[core::modifies(errno)]];
int fscanf(FILE * [[core::noalias]] stream, const char * [[core::noalias]] format, ...)
      [[core::modifies(errno)]];
```

B.17 General utilities <stdlib.h>

`__CORE_VERSION_STDLIB_H__` `EXIT_SUCCESS` `MB_CUR_MAX`
`EXIT_FAILURE` `RAND_MAX`

```
double atof(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
int atoi(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
long int atol(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
long long int atoll(const char *nptr) [[core::modifies(errno), core::evaluates(locale)]];
double strtod(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
float strtof(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
long double strtold(C * [[core::noalias]] nptr, C ** [[core::noalias]] endptr)
      [[core::modifies(errno), core::evaluates(locale)]];
long int strtol(C * [[core::noalias]] nptr,
      C ** [[core::noalias]] endptr,
      int base) [[core::modifies(errno), core::evaluates(locale)]];
long long int strtoll(C * [[core::noalias]] nptr,
      C ** [[core::noalias]] endptr,
      int base)
      [[core::modifies(errno), core::evaluates(locale)]];
unsigned long int strtoul(C * [[core::noalias]] nptr,
      C ** [[core::noalias]] endptr,
      int base)
      [[core::modifies(errno), core::evaluates(locale)]];
unsigned long long int strtoull(C * [[core::noalias]] nptr,
      C ** [[core::noalias]] endptr,
      int base)
      [[core::modifies(errno), core::evaluates(locale)]];
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x)
 [[core::stateless, core::idempotent]];
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x, UT
    baseflags)
 [[core::stateless, core::idempotent]];
size_t totext(size_t n, [[core::noalias(n), core::writethrough]] char s[n], A x, UT
    baseflags,
      const char* [[core::noalias]] glue)
 [[core::stateless, core::idempotent]];
int rand(void) [[core::modifies(rand)]];
void srand(unsigned int seed) [[core::modifies(rand)]];
constexpr void * aligned_alloc(size_t alignment, size_t size)
      [[core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak]];
constexpr void * calloc(size_t nmemb, size_t size)
      [[core::noalias(nmemb × size), core::modifies(malloc), core::noleak]];
constexpr void free(void * [[core::free]] ptr)
      [[core::modifies(malloc), core::noleak]];
constexpr void * malloc(size_t size)
      [[core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak]];
constexpr void * realloc(void * [[core::realloc]] ptr, size_t size)
```

```
      [[ core::noalias(size), core::writethrough, core::modifies(malloc), core::noleak ]] ;
_Noreturn void abort(void)
      [[ core::reentrant, core::modifies(fopen), core::evaluates(stdin, stdout, stderr) ]] ;
int atexit(void (*func)(void))  [[ core::modifies(atexit) ]] ;
int at_quick_exit(void (*func)(void))  [[ core::modifies(at_quick_exit) ]] ;
_Noreturn void exit(int status)
      [[ core::modifies(atexit, fopen), core::evaluates(stdin, stdout, stderr) ]] ;
_Noreturn void _Exit(int status)
      [[ core::reentrant, core::modifies(fopen), core::evaluates(stdin, stdout, stderr) ]] ;
char * [[ core::alias ]] getenv(const char *name)  [[ core::evaluates(environ) ]] ;
_Noreturn void quick_exit(int status)
      [[ core::reentrant, core::modifies(at_quick_exit, fopen), core::evaluates(stdin,
         stdout, stderr) ]] ;
int system(const char *string)  [[ core::modifies(fopen, time) ]] ;
C *bsearch(const void *key, C *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *))
      [[ core::alias(base) ]] ;
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
[[ deprecated("use abs type-generic macro") ]]  int abs(int j);
[[ deprecated("use abs type-generic macro") ]]  long int labs(long int j);
[[ deprecated("use abs type-generic macro") ]]  long long int llabs(long long int j);
constexpr Q div(R numer, S denom);
int mblen(const char *s, size_t n)
      [[ core::evaluates(locale) ]] ;
int mbtowc(wchar_t * [[ core::noalias ]]  pwc, const char * [[ core::noalias ]]  s, size_t n)
      [[ core::evaluates(locale) ]] ;
int wctomb(char *s, wchar_t wc)
      [[ core::evaluates(locale) ]] ;
size_t mbstowcs(wchar_t * [[ core::noalias ]]  pwcs, const char * [[ core::noalias ]]  s, size_t
    n)
      [[ core::evaluates(locale) ]] ;
size_t wcstombs(char * [[ core::noalias ]]  s, const wchar_t * [[ core::noalias ]]  pwcs, size_t
    n)
      [[ core::evaluates(locale) ]] ;
```

B.18 _Noreturn <stdnoreturn.h>

`noreturn`

B.19 String and storage handling <string.h>

`__CORE_VERSION_STDINT_H__`

```
C *memcpy(C * [[ core::noalias, core::writethrough ]]  s1, D * [[ core::noalias ]]  s2, size_t n)
      [[ core::alias(s1) ]] ;
C *memccpy(C * [[ core::noalias, core::writethrough ]]  s1, D * [[ core::noalias ]]  s2, int c,
    size_t n)
      [[ core::alias(s1) ]] ;
C *memmove(C *s1, D *s2, size_t n)
      [[ core::alias(s1) ]] ;
C *strcpy(C * [[ core::noalias, core::writethrough ]]  s1, D * [[ core::noalias ]]  s2)
      [[ core::alias(s1) ]] ;
C *strncpy(C * [[ core::noalias, core::writethrough ]]  s1, D * [[ core::noalias ]]  s2, size_t n)
      [[ core::alias(s1) ]] ;
C *strcat(C * [[ core::noalias ]]  s1, D * [[ core::noalias ]]  s2)
      [[ core::alias(s1) ]] ;
C *strncat(C * [[ core::noalias ]]  s1, D * [[ core::noalias ]]  s2, size_t n)
      [[ core::alias(s1) ]] ;
int memcmp(C *s1, D *s2, size_t n);
```

```
int strcmp(C *s1, D *s2);
int strcoll(C *s1, D *s2)
      [[ core::evaluates(locale) ]] ;
int strncmp(C *s1, D *s2, size_t n);
size_t strxfrm(C * [[ core::noalias, core::writethrough ]]  s1, D * [[ core::noalias ]]  s2,
    size_t n);
C *memchr(C *s, int c, size_t n)
      [[ core::alias(s) ]] ;
C *strchr(C *s, D c) [[ core::alias(s) ]] ;
size_t strcspn(C *s1, D *s2);
C *strpbrk(C *s1, D *s2)
      [[ core::alias(s1) ]] ;
C *strrchr(C *s, D c)
      [[ core::alias(s) ]] ;
size_t strspn(C *s1, D *s2);
C *strstr(C *s1, D *s2)
      [[ core::alias(s1) ]] ;
C *strtok(C * [[ core::noalias ]]  s1, D * [[ core::noalias ]]  s2)
      [[ core::alias(s1) ]] ;
V *tovoidptr(C *s)  [[ core::alias(s) ]] ;
C *memset(C * [[ core::writethrough ]] s, int c, size_t n)  [[ core::alias(s) ]] ;
const char * [[ core::alias ]] strerror(int errnum)  [[ core::evaluates(locale) ]] ;
size_t strlen(A x)
 [[ core::stateless, core::idempotent ]] ;
size_t strlen(A x, UT baseflags)
 [[ core::stateless, core::idempotent ]] ;
size_t strlen(A x, UT baseflags, const char* [[ core::noalias ]] glue)
 [[ core::stateless, core::idempotent ]] ;
char* strdup(A x)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* strdup(A x, UT baseflags)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* strdup(A x, UT baseflags, const char* [[ core::noalias ]] glue)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* [[ core::noalias ]]  strndup(A x, size_t len)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* [[ core::noalias ]]  strndup(A x, size_t len, UT baseflags)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
char* [[ core::noalias ]]  strndup(A x, size_t len, UT baseflags,
const char* [[ core::noalias ]] glue)
      [[ core::noalias, core::writethrough, core::stateless, core::modifies(malloc) ]] ;
```

B.20 Threads `<threads.h>`

`__STDC_NO_THREADS__`	`tss_dtor_t`	`thrd_timedout`
`TSS_DTOR_ITERATIONS`	`thrd_start_t`	`thrd_success`
`cnd_t`	`once_flag`	`thrd_busy`
`thrd_t`	`mtx_plain`	`thrd_error`
`tss_t`	`mtx_recursive`	`thrd_nomem`
`mtx_t`	`mtx_timed`	

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t * [[ core::writethrough ]] cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
      const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

```
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t * [[ core::writethrough ]] mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t * [[ core::noalias, core::writethrough ]] thr, thrd_start_t func, void
     *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
_Noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
void thrd_yield(void);
int tss_create(tss_t * [[ core::writethrough ]] key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

B.21 Date and time <time.h>

`__CORE_VERSION_TIME_H__`	`clock_t`	`struct tm`
`CLOCKS_PER_SEC`	`time_t`	
`TIME_UTC`	`struct timespec`	

```
clock_t clock(void) [[ core::evaluates(time) ]] ;
double difftime(time_t time1, time_t time0);
time_t mktime( [[ core::noalias ]] struct tm timeptr[1]) [[ core::evaluates(time) ]] ;
time_t time(time_t *timer) [[ core::evaluates(time) ]] ;
int timespec_get( [[ core::noalias, core::writethrough ]] struct timespec ts[1], int base)
 [[ core::evaluates(time) ]] ;
char * [[ core::alias ]] asctime( [[ core::noalias ]] const struct tm timeptr[1]);
char *asctime_r( [[ core::noalias ]] const struct tm timeptr[1],
             [[ core::noalias ]] char buf[26])
       [[ core::alias(buf) ]] ;
char *ctime( [[ core::noalias ]] const time_t timer[1])
 [[ core::alias(asctime), core::evaluates(locale, time) ]] ;
char *ctime_r( [[ core::noalias ]] const time_t timer[1], [[ core::noalias, core::writethrough
     ]] char buf[26])
 [[ core::alias(buf), core::evaluates(locale, time) ]] ;
struct tm *gmtime( [[ core::noalias ]] const time_t timer[1])
 [[ core::alias(localtime), core::evaluates(time) ]] ;
struct tm *gmtime_r( [[ core::noalias ]] const time_t timer[1],
       [[ core::noalias, core::writethrough ]] struct tm buf[1])
 [[ core::alias(buf), core::evaluates(time) ]] ;
struct tm * [[ core::alias ]] localtime( [[ core::noalias ]] const time_t timer[1])
 [[ core::evaluates(locale, time) ]] ;
struct tm *localtime_r( [[ core::noalias ]] const time_t timer[1],
       [[ core::noalias, core::writethrough ]] struct tm buf[1])
 [[ core::alias(buf), core::evaluates(locale, time) ]] ;
size_t strftime(char * [[ core::noalias, core::writethrough ]] s, size_t maxsize,
      const char * [[ core::noalias ]] format,
      const struct tm * [[ core::noalias ]] timeptr)
 [[ core::evaluates(locale) ]] ;
```

B.22 Unicode utilities <uchar.h>

__CORE_VERSION_UCHAR_H__ mbstate_t

```
size_t mbrtoc8(char8_t * [[ core::noalias, core::writethrough ]] pc8,
const char * [[ core::noalias ]] s, size_t n, mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]] ;
size_t mbrtoc16(char16_t * [[ core::noalias, core::writethrough ]] pc16,
const char * [[ core::noalias ]] s, size_t n, mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]] ;
size_t c8rtomb(char * [[ core::noalias, core::writethrough ]] s, char8_t c8,
mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]]
size_t c16rtomb(char * [[ core::noalias, core::writethrough ]] s, char16_t c16,
mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]]
size_t mbrtoc32(char32_t * [[ core::noalias, core::writethrough ]] pc32,
const char * [[ core::noalias ]] s, size_t n, mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]]
size_t c32rtomb(char * [[ core::noalias, core::writethrough ]] s, char32_t c32,
mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]]
```

B.23 Extended multibyte/wide character utilities <wchar.h>

__CORE_VERSION_WCHAR_H__
mbstate_t
wint_t
struct tm
WCHAR_MAX
WCHAR_MIN
WEOF

```
int fwprintf(FILE * [[ core::noalias ]] stream, const wchar_t * [[ core::noalias ]] format, ...)
      [[ core::modifies(errno) ]] ;
int fwscanf(FILE * [[ core::noalias ]] stream, const wchar_t * [[ core::noalias ]] format, ...)
      [[ core::modifies(errno) ]] ;
int swprintf(wchar_t * [[ core::noalias ]] s, size_t n, const wchar_t * [[ core::noalias ]]
    format,
     ...) [[ core::modifies(errno) ]] ;
int swscanf(const wchar_t * [[ core::noalias ]] s, const wchar_t * [[ core::noalias ]] format,
    ...)
      [[ core::modifies(errno) ]] ;
int vfwprintf(FILE * [[ core::noalias ]] stream, const wchar_t * [[ core::noalias ]] format,
      va_list arg) [[ core::modifies(errno) ]] ;
int vfwscanf(FILE * [[ core::noalias ]] stream, const wchar_t * [[ core::noalias ]] format,
      va_list arg) [[ core::modifies(errno) ]] ;
int vswprintf(wchar_t * [[ core::noalias ]] s, size_t n, const wchar_t * [[ core::noalias ]]
    format,
      va_list arg) [[ core::modifies(errno) ]] ;
int vswscanf(const wchar_t * [[ core::noalias ]] s, const wchar_t * [[ core::noalias ]] format,
      va_list arg) [[ core::modifies(errno) ]] ;
int vwprintf(const wchar_t * [[ core::noalias ]] format, va_list arg)
      [[ core::modifies(errno), core::evaluates(stdout) ]] ;
int vwscanf(const wchar_t * [[ core::noalias ]] format, va_list arg)
      [[ core::modifies(errno), core::evaluates(stdin) ]] ;
int wprintf(const wchar_t * [[ core::noalias ]] format, ...)
      [[ core::modifies(errno), core::evaluates(stdout) ]] ;
int wscanf(const wchar_t * [[ core::noalias ]] format, ...)
      [[ core::modifies(errno), core::evaluates(stdin) ]] ;
wint_t fgetwc(FILE *stream) [[ core::modifies(errno) ]] ;
wchar_t *fgetws(wchar_t * [[ core::noalias ]] s, int n, FILE * [[ core::noalias ]] stream)
      [[ core::modifies(errno) ]] ;
wint_t fputwc(wchar_t c, FILE *stream) [[ core::modifies(errno) ]] ;
```

```
int fputws(const wchar_t * [[ core::noalias ]] s, FILE * [[ core::noalias ]] stream)
      [[ core::modifies(errno) ]] ;
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void) [[ core::evaluates(stdin) ]] ;
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c) [[ core::evaluates(stdout) ]] ;;
wint_t ungetwc(wint_t c, FILE *stream);
wchar_t *wmemcpy(wchar_t * [[ core::noalias ]] s1, const wchar_t * [[ core::noalias ]] s2,
    size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
size_t wcsftime(wchar_t * [[ core::noalias ]] s, size_t maxsize,
      const wchar_t * [[ core::noalias ]] format, const struct tm * [[ core::noalias ]] timeptr
          )
      [[ core::evaluates(locale) ]] ;
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(locale)
wint_t btowc(int c);
int wctob(wint_t c);
int mbsinit(const mbstate_t *ps);
size_t mbrlen(const char * [[ core::noalias ]] s, size_t n, mbstate_t * [[ core::noalias ]] ps)
    ;
size_t mbrtowc(wchar_t * [[ core::noalias ]] pwc, const char * [[ core::noalias ]] s, size_t n,
      mbstate_t * [[ core::noalias ]] ps) [[ core::modifies(errno) ]] ;
size_t wcrtomb(char * [[ core::noalias ]] s, wchar_t wc, mbstate_t * [[ core::noalias ]] ps)
      [[ core::modifies(errno) ]] ;
size_t mbsrtowcs(wchar_t * [[ core::noalias ]] dst, const char ** [[ core::noalias ]] src,
    size_t len,
      mbstate_t * [[ core::noalias ]] ps) [[ core::modifies(errno) ]] ;
size_t wcsrtombs(char * [[ core::noalias ]] dst, const wchar_t ** [[ core::noalias ]] src,
    size_t len,
      mbstate_t * [[ core::noalias ]] ps) [[ core::modifies(errno) ]] ;
```

B.24 Wide character classification and mapping utilities <wctype.h>

`wint_t` `wctrans_t` `wctype_t` `WEOF`

```
#pragma CORE FUNCTION_ATTRIBUTE core::evaluates(locale)
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
constexpr int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
constexpr int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char *property);
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
```

B.25 Three-way comparison <stdcompare.h>

`std::partial_ordering`
`std::partial_ordering::equivalent`
`std::partial_ordering::greater`
`std::partial_ordering::less`
`std::partial_ordering::unordered`
`std::strong_equality`
`std::strong_equality::equal`
`std::strong_equality::unequal`
`std::strong_ordering`
`std::strong_ordering::equal`
`std::strong_ordering::greater`
`std::strong_ordering::less`
`std::weak_equality`
`std::weak_equality::equivalent`
`std::weak_equality::nonequivalent`
`std::weak_ordering`
`std::weak_ordering::equivalent`
`std::weak_ordering::greater`
`std::weak_ordering::less`

```
bool std::isstrong(R x);
bool std::issymmetric(R x);
S std::toswitch(T x);
S std::tostrong(W x);
Tλ std::tocompare(R x);
R* std::search(size_t n, R base[n], const R key[1]);
void std::sort(size_t n, R base[n]);
```

Annex C
(informative)
Sequence points

1 The following are the sequence points described in 5.1.2.3:

— Between the evaluations of the function designator and actual arguments in a function call and the actual call. (6.5.2.2).

— Between the evaluations of the first and second operands of the following operators: logical AND ~~&&~~ ∧ (6.5.14); logical OR ~~||~~ ∨ (6.5.15); comma `,` (6.5.18).

— Between the evaluations of the first operand of the conditional `?:` operator and whichever of the second and third operands is evaluated (6.5.16).

— Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions: a full declarator for a variably modified type; an initializer that is not part of a compound literal (6.7.12); the expression in an expression statement (6.8.3); the controlling expression of a selection statement (**`if`** or **`switch`**) (6.8.4); the controlling expression of a **`while`** or **`do`** statement (6.8.5); each of the (optional) expressions of a **`for`** statement (6.8.5.3); the (optional) expression in a **`return`** statement (6.8.6.4).

— Immediately before a library function returns (7.1.4).

— After the actions associated with each formatted input/output function conversion specifier (7.21.6, 7.29.2).

— Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call (7.22.6).

Annex D
(normative)
Universal character names for identifiers

1 This clause lists the hexadecimal code values that are valid in universal character names in identifiers.

D.1 Ranges of characters allowed

1 00A8, 00AA, 00AD, 00AF, 00B2–00B5, 00B7–00BA, 00BC–00BE, 00C0–00D6, 00D8–00F6, 00F8–00FF

2 0100–167F, 1681–180D, 180F–1FFF

3 200B–200D, 202A–202E, 203F–2040, 2054, 2060–206F

4 2070–218F, 2460–24FF, 2776–2793, 2C00–2DFF, 2E80–2FFF

5 3004–3007, 3021–302F, 3031–303F

6 3040–D7FF

7 F900–FD3D, FD40–FDCF, FDF0–FE44, FE47–FFFD

8 10000–1FFFD, 20000–2FFFD, 30000–3FFFD, 40000–4FFFD, 50000–5FFFD, 60000–6FFFD, 70000–7FFFD, 80000–8FFFD, 90000–9FFFD, A0000–AFFFD, B0000–BFFFD, C0000–CFFFD, D0000–DFFFD, E0000–EFFFD

D.2 Ranges of characters disallowed initially

1 0300–036F, 1DC0–1DFF, 20D0–20FF, FE20–FE2F

Annex E
(informative)
Implementation limits

1 The contents of the header <limits.h> are given below. The values shall all be constant expressions suitable for use in **#if** preprocessing directives. The components are described further in 5.2.4.2.1.

2 For the following macros, the minimum values shown shall be replaced by implementation-defined values.

```
#define BOOL_WIDTH                1
#define CHAR_BIT                  8
#define USHRT_WIDTH              16
#define UINT_WIDTH               16
#define ULONG_WIDTH              32
#define ULLONG_WIDTH             64
#define INT_BITFIELD_MAX         UINT_WIDTH
#define MB_LEN_MAX                1
```

3 For the following macros, the minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign that are deduced from the macros above as indicated.[464]

```
#define BOOL_MAX                      1  // 2^{BOOL_WIDTH} - 1
#define CHAR_MAX     UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN             0 or SCHAR_MIN
#define CHAR_WIDTH                    8  // CHAR_BIT
#define INT_MAX                  +32767  // 2^{INT_WIDTH-1} - 1
#define INT_MIN                  -32768  // -2^{INT_WIDTH-1}
#define INT_WIDTH                    16  // UINT_WIDTH
#define LONG_MAX            +2147483647  // 2^{LONG_WIDTH-1} - 1
#define LONG_MIN            -2147483648  // -2^{LONG_WIDTH-1}
#define LONG_WIDTH                   32  // ULONG_WIDTH
#define LLONG_MAX  +9223372036854775807  // 2^{LLONG_WIDTH-1} - 1
#define LLONG_MIN  -9223372036854775808  // -2^{LLONG_WIDTH-1}
#define LLONG_WIDTH                  64  // ULLONG_WIDTH
#define SCHAR_MAX                  +127  // 2^{SCHAR_WIDTH-1} - 1
#define SCHAR_MIN                  -128  // -2^{SCHAR_WIDTH-1}
#define SCHAR_WIDTH                   8  // CHAR_BIT
#define SHRT_MAX                 +32767  // 2^{SHRT_WIDTH-1} - 1
#define SHRT_MIN                 -32768  // -2^{SHRT_WIDTH-1}
#define UCHAR_MAX                   255  // 2^{UCHAR_WIDTH} - 1
#define UCHAR_WIDTH                   8  // CHAR_BIT
#define USHRT_MAX                 65535  // 2^{USHRT_WIDTH} - 1
#define UINT_MAX                  65535  // 2^{UINT_WIDTH} - 1
#define ULONG_MAX            4294967295  // 2^{ULONG_WIDTH} - 1
#define ULLONG_MAX 18446744073709551615  // 2^{ULLONG_WIDTH} - 1
```

4 The contents of the header <float.h> are given below. All integer values, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. The components are described further in 5.2.4.2.2.

5 The values given in the following list shall be replaced by implementation-defined expressions:

```
#define FLT_EVAL_METHOD
#define FLT_ROUNDS
```

[464] For the minimum value of a signed integer type there is no expression consisting of a minus sign and a decimal literal of that same type. The numbers in the table are only given as indications for the values and do not represent suitable expressions to be used for these macros.

6 The values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

```
#define DBL_DECIMAL_DIG          10
#define DBL_DIG                  10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP          +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP          -37
#define DBL_MIN_EXP
#define DECIMAL_DIG              10
#define FLT_DECIMAL_DIG           6
#define FLT_DIG                   6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP          +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP          -37
#define FLT_MIN_EXP
#define FLT_RADIX                 2
#define LDBL_DECIMAL_DIG         10
#define LDBL_DIG                 10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP         +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP         -37
#define LDBL_MIN_EXP
```

7 The values given in the following list shall be replaced by implementation-defined constant expressions with values that are greater than or equal to those shown:

```
#define DBL_MAX                1E+37
#define DBL_NORM_MAX           1E+37
#define FLT_MAX                1E+37
#define FLT_NORM_MAX           1E+37
#define LDBL_MAX               1E+37
#define LDBL_NORM_MAX          1E+37
```

8 The values given in the following list shall be replaced by implementation-defined constant expressions with (positive) values that are less than or equal to those shown:

```
#define DBL_EPSILON             1E-9
#define DBL_MIN                1E-37
#define FLT_EPSILON             1E-5
#define FLT_MIN                1E-37
#define LDBL_EPSILON            1E-9
#define LDBL_MIN               1E-37
```

Annex F

(normative)

IEC 60559 floating-point arithmetic

F.1 Introduction

1 This annex specifies C language support for the IEC 60559 floating-point standard. The *IEC 60559 floating-point standard* is specifically *Binary floating-point arithmetic for microprocessor systems, second edition* (IEC 60559:1989), previously designated IEC 559:1989 and as *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE 754–1985). *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854–1987) generalizes the binary standard to remove dependencies on radix and word length. *IEC 60559* generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc. An implementation that defines **`__STDC_IEC_559__`** shall conform to the specifications in this annex.[465] Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless explicitly stated otherwise. Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values.

F.2 Types

1 The C floating types match the IEC 60559 formats as follows:

— The **`float`** type matches the IEC 60559 single format.

— The **`double`** type matches the IEC 60559 double format.

— The **`long double`** type matches an IEC 60559 extended format,[466] else a non-IEC 60559 extended format, else the IEC 60559 **`double`** format.

Any non-IEC 60559 extended format used for the **`long double`** type shall have more precision than IEC 60559 double and at least the range of IEC 60559 double.[467] The value of **`FLT_ROUNDS`** applies to all IEC 60559 types supported by the implementation, but need not apply to non-IEC 60559 types.

Recommended practice

2 The **`long double`** type should match an IEC 60559 extended format.

F.2.1 Infinities, signed zeros, and NaNs

1 This specification does not define the behavior of signaling NaNs.[468] It generally uses the term *NaN* to denote quiet NaNs. The **`NAN`** and **`INFINITY`** macros and the **`nan`** functions in `<math.h>` provide designations for IEC 60559 NaNs and infinities.

F.3 Operators and functions

1 C operators and functions provide IEC 60559 required and recommended facilities as listed below.

— The `+`, `-`, ~~`*`~~×, and `/` operators provide the IEC 60559 add, subtract, multiply, and divide operations.

— The **`sqrt`** functions in `<math.h>` provide the IEC 60559 square root operation.

— The **`remainder`** functions in `<math.h>` provide the IEC 60559 remainder operation. The **`remquo`** functions in `<math.h>` provide the same operation but with additional information.

[465] Implementations that do not define **`__STDC_IEC_559__`** are not required to conform to these specifications.

[466] "Extended" is IEC 60559's double-extended data format. Extended refers to both the common 80-bit and quadruple 128-bit IEC 60559 formats.

[467] A non-IEC 60559 **`long double`** type is required to provide infinity and NaNs, as its values include all **`double`** values.

[468] Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

— The **rint** functions in <math.h> provide the IEC 60559 operation that rounds a floating-point number to an integer value (in the same precision). The **nearbyint** functions in <math.h> provide the nearbyinteger function recommended in the Appendix to ANSI/IEEE 854.

— The conversions for floating types provide the IEC 60559 conversions between floating-point precisions.

— The conversions from integer to floating types provide the IEC 60559 conversions from integer to floating point.

— The conversions from floating to integer types provide IEC 60559-like conversions but always round toward zero.

— The **lrint** and **llrint** functions in <math.h> provide the IEC 60559 conversions, which honor the directed rounding mode, from floating point to the **long int** and **long long int** integer formats. The **lrint** and **llrint** functions can be used to implement IEC 60559 conversions from floating to other integer formats.

— The translation time conversion of floating constants and the **strtod**, **strtof**, **strtold**, **fprintf**, **fscanf**, and related library functions in <stdlib.h>, <stdio.h>, and <wchar.h> provide IEC 60559 binary-decimal conversions. The **strtold** function in <stdlib.h> provides the conv function recommended in the Appendix to ANSI/IEEE 854.

— The relational and equality operators provide IEC 60559 comparisons. IEC 60559 identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison macros (**isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, and **isunordered**) in <math.h> supplement the language operators to address this need. The **islessgreater** and **isunordered** macros provide respectively a quiet version of the <> predicate and the unordered predicate recommended in the Appendix to IEC 60559.

— The **feclearexcept**, **feraiseexcept**, and **fetestexcept** functions in <fenv.h> provide the facility to test and alter the IEC 60559 floating-point exception status flags. The **fegetexceptflag** and **fesetexceptflag** functions in <fenv.h> provide the facility to save and restore all five status flags at one time. These functions are used in conjunction with the type **fexcept_t** and the floating-point exception macros (**FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, **FE_INVALID**) also in <fenv.h>.

— The **fegetround** and **fesetround** functions in <fenv.h> provide the facility to select among the IEC 60559 directed rounding modes represented by the rounding direction macros in <fenv.h> (**FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, **FE_TOWARDZERO**), **FE_TONEARESTFROMZERO** and the values 0, 1, 2, and 3 of **FLT_ROUNDS** are the IEC 60559 directed rounding modes.

— The **fegetenv**, **feholdexcept**, **fesetenv**, and **feupdateenv** functions in <fenv.h> provide a facility to manage the floating-point environment, comprising the IEC 60559 status flags and control modes.

— The **copysign** functions in <math.h> provide the copysign function recommended in the Appendix to IEC 60559.

— The **fabs** functions in <math.h> provide the abs function recommended in the Appendix to IEC 60559.

— The unary minus (-) operator provides the unary minus (-) operation recommended in the Appendix to IEC 60559.

— The **scalbn** and **scalbln** functions in <math.h> provide the scalb function recommended in the Appendix to IEC 60559.

— The **logb** functions in <math.h> provide the logb function recommended in the Appendix to IEC 60559, but following the newer specifications in ANSI/IEEE 854.

— The **nextafter** and **nexttoward** functions in <math.h> provide the nextafter function recommended in the Appendix to IEC 60559 (but with a minor change to better handle signed zeros).

— The **isfinite** macro in <math.h> provides the finite function recommended in the Appendix to IEC 60559.

— The **isnan** macro in <math.h> provides the isnan function recommended in the Appendix to IEC 60559.

— The **signbit** macro and the **fpclassify** macro in <math.h>, used in conjunction with the number classification macros (**FP_NAN**, **FP_INFINITE**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO**), provide the facility of the class function recommended in the Appendix to IEC 60559 (except that the classification macros defined in 7.12.3 do not distinguish signaling from quiet NaNs).

F.4 Floating to integer conversion

1 If the integer type is **bool** , 6.3.1.2 applies and no floating-point exceptions are raised (even for NaN). Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the "invalid" floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the "inexact" floating-point exception is unspecified.[469)]

F.5 Binary-decimal conversion

1 Conversions involving IEC 60559 formats follow all pertinent recommended practice. Conversion between any supported IEC 60559 format and decimal character sequence with M or fewer significant digits is correctly rounded (honoring the current rounding mode), where M is the maximum value of the *T*_**DECIMAL_DIG** macros (defined in <float.h>). Conversion from any supported IEC 60559 format to decimal character sequence with at least *T*_**DECIMAL_DIG** digits (for the corresponding type) and back, using to-nearest rounding, is the identity function.

2 Functions such as **strtod** that convert character sequences to floating types honor the rounding direction. Hence, if the rounding direction might be upward or downward, the implementation cannot convert a minus-signed sequence by negating the converted unsigned sequence.

3 **NOTE** IEC 60559 specifies that conversion to one-digit character strings using roundTiesToEven when both choices have an odd least significant digit, shall produce the value with the larger magnitude. For example, this can happen with `9.5e2` whose nearest neighbors are `9.e2` and `1.e3`, both of which have a single odd digit in the significand part.

F.6 The `return` statement

If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment[470)] to the return type of the function and the resulting value is returned to the caller.

F.7 Contracted expressions

1 A contracted expression is correctly rounded (once) and treats infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by IEC 60559.

Recommended practice

2 A contracted expression should raise floating-point exceptions in a manner generally consistent with the basic arithmetic operations.

[469)] ANSI/IEEE 854, but not IEC 60559 (ANSI/IEEE 754), directly specifies that floating-to-integer conversions raise the "inexact" floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the "inexact" floating-point exception. See **rint**, **lrint**, **llrint**, and **nearbyint** in <math.h>.

[470)] Assignment removes any extra range and precision.

F.8 Floating-point environment

1 The floating-point environment defined in <fenv.h> includes the IEC 60559 floating-point exception status flags and directed-rounding control modes. It includes also IEC 60559 dynamic rounding precision and trap enablement modes, if the implementation supports them.[471)]

F.8.1 Environment management

1 IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. When the state for the **FENV_ACCESS** pragma (defined in <fenv.h>) is "on", these changes to the floating-point state are treated as side effects which respect sequence points.[472)]

F.8.2 Translation

1 During translation the IEC 60559 default modes are in effect:

— The rounding direction mode is rounding to nearest.

— The rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

Recommended practice

2 The implementation should produce a diagnostic message for each translation-time floating-point exception, other than "inexact";[473)] the implementation should then proceed with the translation of the program.

F.8.3 Execution

1 At program startup the floating-point environment is initialized as prescribed by IEC 60559:

— All floating-point exception status flags are cleared.

— The rounding direction mode is rounding to nearest.

— The dynamic rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

F.8.4 Constant expressions

1 An arithmetic constant expression of floating type, other than one in an initializer for an object that has static or thread storage duration, is evaluated (as if) during execution; thus, it is affected by any operative floating-point control modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is "on").[474)]

[471)] This specification does not require dynamic rounding precision nor trap enablement modes.

[472)] If the state for the **FENV_ACCESS** pragma is "off", the implementation is free to assume the floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see F.9).

[473)] As floating constants are converted to appropriate internal representations at translation time, their conversion is subject to default rounding modes and raises no execution-time floating-point exceptions (even where the state of the **FENV_ACCESS** pragma is "on"). Library functions, for example **strtod**, provide execution-time conversion of numeric strings.

[474)] Where the state for the **FENV_ACCESS** pragma is "on", results of inexact expressions like `1.0/3.0` are affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` generate execution-time floating-point exceptions. The programmer can achieve the efficiency of translation-time evaluation through static initialization, such as

```
const static double one_third = 1.0/3.0;
```

2 **EXAMPLE**

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
      float w[] = { 0.0/0.0 };  // raises an exception
      static float x = 0.0/0.0; // does not raise an exception
      float y = 0.0/0.0;        // raises an exception
      double z = 0.0/0.0;       // raises an exception
      /* ... */
}
```

3 For the static initialization, the division is done at translation time, raising no (execution-time) floating-point exceptions. On the other hand, for the three automatic initializations the invalid division occurs at execution time.

F.8.5 Initialization

1 All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is "on"). All computation for initialization of objects that have static or thread storage duration is done (as if) at translation time.

2 **EXAMPLE**

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
      float u[] = { 1.1e75 };  // raises exceptions
      static float v = 1.1e75; // does not raise exceptions
      float w = 1.1e75;        // raises exceptions
      double x = 1.1e75;       // may raise exceptions
      float y = 1.1e75f;       // may raise exceptions
      long double z = 1.1e75;  // does not raise exceptions
      /* ... */
}
```

3 The static initialization of `v` raises no (execution-time) floating-point exceptions because its computation is done at translation time. The automatic initialization of `u` and `w` require an execution-time conversion to **`float`** of the wider value `1.1e75`, which raises floating-point exceptions. The automatic initializations of `x` and `y` entail execution-time conversion; however, in some expression evaluation methods, the conversions is not to a narrower format, in which case no floating-point exception is raised.[475] The automatic initialization of `z` entails execution-time conversion, but not to a narrower format, so no floating-point exception is raised. Note that the conversions of the floating constants `1.1e75` and `1.1e75f` to their internal representations occur at translation time in all cases.

F.8.6 Changing the environment

1 Operations defined in 6.5 and functions and macros defined for the standard libraries change floating-point status flags and control modes just as indicated by their specifications (including conformance to IEC 60559). They do not change flags or modes (so as to be detectable by the user) in any other cases.

2 If the argument to the **`feraiseexcept`** function in `<fenv.h>` represents IEC 60559 valid coincident floating-point exceptions for atomic operations (namely "overflow" and "inexact", or "underflow" and "inexact"), then "overflow" or "underflow" is raised before "inexact".

[475]Use of **`float_t`** and **`double_t`** variables increases the likelihood of translation-time computation. For example, the automatic initialization

```
double_t x = 1.1e75;
```

could be done at translation time, regardless of the expression evaluation method.

F.9 Optimization

1 This section identifies code transformations that might subvert IEC 60559-specified behavior, and others that do not.

F.9.1 Global transformations

1 Floating-point arithmetic operations and external function calls may entail side effects which optimization shall honor, at least where the state of the `FENV_ACCESS` pragma is "on". The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations (`+`, `*`, etc.) implicitly read the modes and write the flags.

2 Concern about side effects may inhibit code motion and removal of seemingly useless code. For example, in

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(double x)
{
      /* ... */
      for (i = 0; i  <  n; i++) x + 1;
      /* ... */
}
```

`x+1` might raise floating-point exceptions, so cannot be removed. And since the loop body might not execute (maybe $0 \geq$ `n`), `x+1` cannot be moved out of the loop. (Of course these optimizations are valid if the implementation can rule out the nettlesome cases.)

3 This specification does not require support for trap handlers that maintain information about the order or count of floating-point exceptions. Therefore, between function calls, floating-point exceptions need not be precise: the actual order and number of occurrences of floating-point exceptions (> 1) may vary from what the source code expresses. Thus, the preceding loop could be treated as

```
if (0  <  n) x + 1;
```

F.9.2 Expression transformations

1 $x/2 \leftrightarrow x \times 0.5$ Although similar transformations involving inexact constants generally do not yield numerically equivalent expressions, if the constants are exact then such transformations can be made on IEC 60559 machines and others that round perfectly.

$1 \times x$ and $x/1 \to x$ The expressions $1 \times x$, $x/1$, and x are equivalent (on IEC 60559 machines, among others).[476)]

$x/x \to 1.0$ The expressions x/x and 1.0 are not equivalent if x can be zero, infinite, or NaN.

$x - y \leftrightarrow x + (-y)$ The expressions $x - y$, $x + (-y)$, and $(-y) + x$ are equivalent (on IEC 60559 machines, among others).

$x - y \leftrightarrow -(y - x)$ The expressions $x - y$ and $-(y - x)$ are not equivalent because $1 - 1$ is $+0$ but $-(1 - 1)$ is -0 (in the default rounding direction).[477)]

$x - x \to 0.0$ The expressions $x - x$ and 0.0 are not equivalent if x is a NaN or infinite.

476)Strict support for signaling NaNs — not required by this specification — would invalidate these and other transformations that remove arithmetic operators.

477)IEC 60559 prescribes a signed zero to preserve mathematical identities across certain discontinuities. Examples include:

$1/(1/\pm\infty)$ is $\pm\infty$

and

conj(csqrt(z)) is csqrt(conj(z)),

for complex z.

$0 \times x \to 0.0$ The expressions $0 \times x$ and 0.0 are not equivalent if x is a NaN, infinite, or -0.

$x + 0 \to x$ The expressions $x + 0$ and x are not equivalent if x is -0, because $(-0) + (+0)$ yields $+0$ (in the default rounding direction), not -0.

$x - 0 \to x$ $(+0) - (+0)$ yields -0 when rounding is downward (toward $-\infty$), but $+0$ otherwise, and $(-0) - (+0)$ always yields -0; so, if the state of the `FENV_ACCESS` pragma is "off", promising default rounding, then the implementation can replace $x - 0$ by x, even if x might be zero.

$-x \leftrightarrow 0 - x$ The expressions $-x$ and $0 - x$ are not equivalent if x is $+0$, because $-(+0)$ yields -0, but $0 - (+0)$ yields $+0$ (unless rounding is downward).

F.9.3 Relational operators

1 $x \neq x \to$ **`false`** The expression $x \neq x$ is true if x is a NaN.

$x = x \to$ **`true`** The expression $x = x$ is false if x is a NaN.

$x < y \to$ **`isless`**(x, y) (and similarly for $\leq$, $>$, $\geq$) Though numerically equal, these expressions are not equivalent because of side effects when x or y is a NaN and the state of the `FENV_ACCESS` pragma is "on". This transformation, which would be desirable if extra code were required to cause the "invalid" floating-point exception for unordered cases, could be performed provided the state of the `FENV_ACCESS` pragma is "off".

The sense of relational operators shall be maintained. This includes handling unordered cases as expressed by the source code.

2 **EXAMPLE**

```
// calls g and raises “invalid” if a and b are unordered
if (a  <  b)
      f();
else
      g();
```

is not equivalent to

```
// calls f and raises “invalid” if a and b are unordered
if (a >= b)
if (a ≥ b)
      g();
else
      f();
```

nor to

```
// calls f without raising “invalid” if a and b are unordered
if (isgreaterequal(a,b))
      g();
else
      f();
```

nor, unless the state of the `FENV_ACCESS` pragma is "off", to

```
// calls g without raising “invalid” if a and b are unordered
if (isless(a,b))
      f();
else
      g();
```

but is equivalent to

```
    if (!(a < b))
    if (¬(a < b))
        g();
    else
        f();
```

F.9.4 Constant arithmetic

1 The implementation shall honor floating-point exceptions raised by execution-time constant arithmetic wherever the state of the **FENV_ACCESS** pragma is "on". (See F.8.4 and F.8.5.) An operation on constants that raises no floating-point exception can be folded during translation, except, if the state of the **FENV_ACCESS** pragma is "on", a further check is required to assure that changing the rounding direction to downward does not alter the sign of the result,[478] and implementations that support dynamic rounding precision modes shall assure further that the result of the operation raises no floating-point exception when converted to the semantic type of the operation.

F.10 Mathematics `<math.h>`

1 This subclause contains specifications of `<math.h>` facilities that are particularly suited for IEC 60559 implementations.

2 The Standard C macro **HUGE_VAL** and its **float** and **long double** analogs, **HUGE_VALF** and **HUGE_VALL**, expand to expressions whose values are positive infinities.

3 Special cases for functions in `<math.h>` are covered directly or indirectly by IEC 60559. The functions that IEC 60559 specifies directly are identified in F.3. The other functions in `<math.h>` treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the **FENV_ACCESS** pragma is "on") the floating-point status flags in a manner consistent with the basic arithmetic operations covered by IEC 60559.

4 The expression **math_errhandling** `&` **MATH_ERREXCEPT** shall evaluate to a nonzero value.

5 The "invalid" and "divide-by-zero" floating-point exceptions are raised as specified in subsequent subclauses of this annex.

6 The "overflow" floating-point exception is raised whenever an infinity — or, because of rounding direction, a maximal-magnitude finite number — is returned in lieu of a value whose magnitude is too large.

7 The "underflow" floating-point exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy.[479]

8 Whether or when library functions raise the "inexact" floating-point exception is unspecified, unless explicitly specified otherwise.

9 Whether or when library functions raise an undeserved "underflow" floating-point exception is unspecified.[480] Otherwise, as implied by F.8.6, the `<math.h>` functions do not raise spurious floating-point exceptions (detectable by the user), other than the "inexact" floating-point exception.

10 Whether the functions honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

11 Functions with a NaN argument return a NaN result and raise no floating-point exception, except where explicitly stated otherwise.

12 The specifications in the following subclauses append to the definitions in `<math.h>`. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol "±" occurs in both an argument and the result, the result has the same sign as the argument.

[478] `0-0` yields `-0` instead of `+0` just when the rounding direction is downward.

[479] IEC 60559 allows different definitions of underflow. They all result in the same values, but differ on when the floating-point exception is raised.

[480] It is intended that undeserved "underflow" and "inexact" floating-point exceptions are raised only if avoiding them would be too costly.

Recommended practice

13 If a function with one or more NaN arguments returns a NaN result, the result should be the same as one of the NaN arguments (after possible type conversion), except perhaps for the sign.

F.10.1 Trigonometric functions

F.10.1.1 The `acos` type-generic macro

1 — `acos`(1) returns +0.

— `acos`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.2 The `asin` type-generic macro

1 — `asin`(± 0) returns ± 0.

— `asin`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.3 The `atan` type-generic macro

1 — `atan`(± 0) returns ± 0.

— `atan`($\pm\infty$) returns $\pm\frac{\pi}{2}$.

F.10.1.4 The `atan2` type-generic macro

1 — `atan2`($\pm 0, -0$) returns $\pm\pi$.[481]

— `atan2`($\pm 0, +0$) returns ± 0.

— `atan2`($\pm 0, x$) returns $\pm\pi$ for $x < 0$.

— `atan2`($\pm 0, x$) returns ± 0 for $x > 0$.

— `atan2`($y, \pm 0$) returns $-\frac{\pi}{2}$ for $y < 0$.

— `atan2`($y, \pm 0$) returns $\frac{\pi}{2}$ for $y > 0$.

— `atan2`($\pm y, -\infty$) returns $\pm\pi$ for finite $y > 0$.

— `atan2`($\pm y, +\infty$) returns ± 0 for finite $y > 0$.

— `atan2`($\pm\infty, x$) returns $\pm\frac{\pi}{2}$ for finite x.

— `atan2`($\pm\infty, -\infty$) returns $\pm\frac{3\pi}{4}$.

— `atan2`($\pm\infty, +\infty$) returns $\pm\frac{\pi}{4}$.

F.10.1.5 The `cos` type-generic macro

1 — `cos`(± 0) returns 1.

— `cos`($\pm\infty$) returns a NaN and raises the "invalid" floating-point exception.

F.10.1.6 The `sin` type-generic macro

1 — `sin`(± 0) returns ± 0.

— `sin`($\pm\infty$) returns a NaN and raises the "invalid" floating-point exception.

[481] `atan2`$(0, 0)$ does not raise the "invalid" floating-point exception, nor does `atan2`$(y, 0)$ raise the "divide-by-zero" floating-point exception.

F.10.1.7 The `tan` type-generic macro

1 — `tan`(± 0) returns ± 0.

— `tan`$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.2 Hyperbolic functions

F.10.2.1 The `acosh` type-generic macro

1 — `acosh`(1) returns $+0$.

— `acosh`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 1$.

— `acosh`$(+\infty)$ returns $+\infty$.

F.10.2.2 The `asinh` type-generic macro

1 — `asinh`(± 0) returns ± 0.

— `asinh`$(\pm\infty)$ returns $\pm\infty$.

F.10.2.3 The `atanh` type-generic macro

1 — `atanh`(± 0) returns ± 0.

— `atanh`(± 1) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception.

— `atanh`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.2.4 The `cosh` type-generic macro

1 — `cosh`(± 0) returns 1.

— `cosh`$(\pm\infty)$ returns $+\infty$.

F.10.2.5 The `sinh` type-generic macro

1 — `sinh`(± 0) returns ± 0.

— `sinh`$(\pm\infty)$ returns $\pm\infty$.

F.10.2.6 The `tanh` type-generic macro

1 — `tanh`(± 0) returns ± 0.

— `tanh`$(\pm\infty)$ returns ± 1.

F.10.3 Exponential and logarithmic functions

F.10.3.1 The `exp` type-generic macro

1 — `exp`(± 0) returns 1.

— `exp`$(-\infty)$ returns $+0$.

— `exp`$(+\infty)$ returns $+\infty$.

F.10.3.2 The `exp2` type-generic macro

1 — `exp2`(± 0) returns 1.

— `exp2`$(-\infty)$ returns $+0$.

— `exp2`$(+\infty)$ returns $+\infty$.

F.10.3.3 The `expm1` type-generic macro

1
- `expm1`(± 0) returns ± 0.
- `expm1`$(-\infty)$ returns -1.
- `expm1`$(+\infty)$ returns $+\infty$.

F.10.3.4 The `frexp` type-generic macro

1
- `frexp`$(\pm 0, exp)$ returns ± 0, and stores 0 in the object pointed to by `exp`.
- `frexp`$(\pm\infty, exp)$ returns $\pm\infty$, and stores an unspecified value in the object pointed to by `exp`.
- `frexp`(NaN, exp) stores an unspecified value in the object pointed to by `exp` (and returns a NaN).

2 `frexp` raises no floating-point exceptions.

3 When the radix of the argument is a power of 2, the returned value is exact and is independent of the current rounding direction mode.

4 On a binary system, the body of the `frexp` function might be

```
{
        *exp = (value == 0) ? 0: (int)(1 + logb(value));
        return scalbn(value, -(*exp));
}
```

F.10.3.5 The `ilogb` type-generic macro

1 When the correct result is representable in the range of the return type, the returned value is exact and is independent of the current rounding direction mode.

2 If the correct result is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised.

3 `ilogb`(x), for x zero, infinite, or NaN, raises the "invalid" floating-point exception and returns the value specified in 7.12.6.5.

F.10.3.6 The `ldexp` type-generic macro

1 On a binary system, `ldexp(x, exp)` is equivalent to `scalbn(x, exp)`.

F.10.3.7 The `log` type-generic macro

1
- `log`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log`(1) returns +0.
- `log`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- `log`$(+\infty)$ returns $+\infty$.

F.10.3.8 The `log10` type-generic macro

1
- `log10`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log10`(1) returns $+0$.
- `log10`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- `log10`$(+\infty)$ returns $+\infty$.

F.10.3.9 The `log1p` type-generic macro

1 — `log1p`(± 0) returns ± 0.

— `log1p`(-1) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.

— `log1p`(x) returns a NaN and raises the "invalid" floating-point exception for $x < -1$.

— `log1p`$(+\infty)$ returns $+\infty$.

F.10.3.10 The `log2` type-generic macro

1 — `log2`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.

— `log2`(1) returns $+0$.

— `log2`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.

— `log2`$(+\infty)$ returns $+\infty$.

F.10.3.11 The `logb` type-generic macro

1 — `logb`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.

— `logb`$(\pm\infty)$ returns $+\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.3.12 The `modf` functions

1 — `modf`$(\pm x, iptr)$ returns a result with the same sign as x.

— `modf`$(\pm\infty, iptr)$ returns ± 0 and stores $\pm\infty$ in the object pointed to by `iptr`.

— `modf`(NaN, iptr) stores a NaN in the object pointed to by `iptr` (and returns a NaN).

2 The returned values are exact and are independent of the current rounding direction mode.

3 `modf` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double modf(double value, double *iptr)
{
      int save_round = fegetround();
      fesetround(FE_TOWARDZERO);
      *iptr = nearbyint(value);
      fesetround(save_round);
      return copysign(
            isinf(value) ? 0.0:
                  value - (*iptr), value);
}
```

F.10.3.13 The `scalbn` and `scalbln` type-generic macros

1 — `scalbn`$(\pm 0, n)$ returns ± 0.

— `scalbn`$(x, 0)$ returns x.

— `scalbn`$(\pm\infty, n)$ returns $\pm\infty$.

2 If the calculation does not overflow or underflow, the returned value is exact and independent of the current rounding direction mode.

F.10.4 Power and absolute value functions

F.10.4.1 The `cbrt` type-generic macro

1 — `cbrt`(± 0) returns ± 0.

— `cbrt`$(\pm\infty)$ returns $\pm\infty$.

F.10.4.2 The `fabs` type-generic macro

1 — `fabs`(± 0) returns $+0$.

— `fabs`$(\pm\infty)$ returns $+\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.4.3 The `hypot` type-generic macro

1 — `hypot`(x, y), `hypot`(y, x), and `hypot`$(x, -y)$ are equivalent.

— `hypot`$(x, \pm 0)$ is equivalent to `fabs`(x).

— `hypot`$(\pm\infty, y)$ returns $+\infty$, even if y is a NaN.

F.10.4.4 The `pow` type-generic macro

1 — `pow`$(\pm 0, y)$ returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for y an odd integer < 0.

— `pow`$(\pm 0, y)$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception for $y < 0$, finite, and not an odd integer.

— `pow`$(\pm 0, -\infty)$ returns $+\infty$.

— `pow`$(\pm 0, y)$ returns ± 0 for y an odd integer > 0.

— `pow`$(\pm 0, y)$ returns $+0$ for $y > 0$ and not an odd integer.

— `pow`$(-1, \pm\infty)$ returns 1.

— `pow`$(+1, y)$ returns 1 for any y, even a NaN.

— `pow`$(x, \pm 0)$ returns 1 for any x, even a NaN.

— `pow`(x, y) returns a NaN and raises the "invalid" floating-point exception for finite $x < 0$ and finite non-integer y.

— `pow`$(x, -\infty)$ returns $+\infty$ for $|x| < 1$.

— `pow`$(x, -\infty)$ returns $+0$ for $|x| > 1$.

— `pow`$(x, +\infty)$ returns $+0$ for $|x| < 1$.

— `pow`$(x, +\infty)$ returns $+\infty$ for $|x| > 1$.

— `pow`$(-\infty, y)$ returns -0 for y an odd integer < 0.

— `pow`$(-\infty, y)$ returns $+0$ for $y < 0$ and not an odd integer.

— `pow`$(-\infty, y)$ returns $-\infty$ for y an odd integer > 0.

— `pow`$(-\infty, y)$ returns $+\infty$ for $y > 0$ and not an odd integer.

— `pow`$(+\infty, y)$ returns $+0$ for $y < 0$.

— `pow`$(+\infty, y)$ returns $+\infty$ for $y > 0$.

F.10.4.5 The `sqrt` type-generic macro

1 `sqrt` is fully specified as a basic arithmetic operation in IEC 60559. The returned value is dependent on the current rounding direction mode.

F.10.5 Error and gamma functions

F.10.5.1 The `erf` type-generic macro

1 — `erf`(± 0) returns ± 0.

— `erf`$(\pm\infty)$ returns ± 1.

F.10.5.2 The `erfc` type-generic macro

1 — `erfc`$(-\infty)$ returns 2.

— `erfc`$(+\infty)$ returns $+0$.

F.10.5.3 The `lgamma` type-generic macro

1 — `lgamma`(1) returns +0.

— `lgamma`(2) returns +0.

— `lgamma`(x) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for x a negative integer or zero.

— `lgamma`$(-\infty)$ returns $+\infty$.

— `lgamma`$(+\infty)$ returns $+\infty$.

F.10.5.4 The `tgamma` type-generic macro

1 — `tgamma`(± 0) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception.

— `tgamma`(x) returns a NaN and raises the "invalid" floating-point exception for x a negative integer.

— `tgamma`$(-\infty)$ returns a NaN and raises the "invalid" floating-point exception.

— `tgamma`$(+\infty)$ returns $+\infty$.

F.10.6 Nearest integer functions

F.10.6.1 The `ceil` type-generic macro

1 — `ceil`(± 0) returns ± 0.

— `ceil`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is independent of the current rounding direction mode.

3 The **double** version of `ceil` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double ceil(double x)
{
      double result;
      int save_round = fegetround();
      fesetround(FE_UPWARD);
      result = rint(x); // or nearbyint instead of rint
      fesetround(save_round);
      return result;
}
```

4 The `ceil` type-generic macro may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments, as this implementation does.

F.10.6.2 The floor type-generic macro

1 — `floor`(± 0) returns ± 0.

— `floor`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value and is independent of the current rounding direction mode.

3 See the sample implementation for `ceil` in F.10.6.1 . The `floor` type-generic macro may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments, as that implementation does.

F.10.6.3 The nearbyint type-generic macro

1 The `nearbyint` functions use IEC 60559 rounding according to the current rounding direction. They do not raise the "inexact" floating-point exception if the result differs in value from the argument.

— `nearbyint`(± 0) returns ± 0 (for all rounding directions).

— `nearbyint`$(\pm\infty)$ returns $\pm\infty$ (for all rounding directions).

F.10.6.4 The rint type-generic macro

1 The `rint` functions differ from the `nearbyint` functions only in that they do raise the "inexact" floating-point exception if the result differs in value from the argument.

F.10.6.5 The lrint and llrint functions

1 The `lrint` and `llrint` functions provide floating-to-integer conversion as prescribed by IEC 60559. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the "inexact" floating-point exception.

F.10.6.6 The round type-generic macro

1 — `round`(± 0) returns ± 0.

— `round`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is independent of the current rounding direction mode.

3 The `double` version of `round` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double round(double x)
{
      double result;
      fenv_t save_env;
      feholdexcept(&save_env);
      result = rint(x);
      if (fetestexcept(FE_INEXACT)) {
            fesetround(FE_TOWARDZERO);
            result = rint(copysign(0.5 + fabs(x), x));
      }
      feupdateenv(&save_env);
      return result;
}
```

The `round` functions may, but are not required to, raise the "inexact" floating-point exception for finite non-integer numeric arguments, as this implementation does.

F.10.6.7 The lround and llround functions

1 The **lround** and **llround** functions differ from the **lrint** and **llrint** functions with the default rounding direction just in that the **lround** and **llround** functions round halfway cases away from zero and need not raise the "inexact" floating-point exception for non-integer arguments that round to within the range of the return type.

F.10.6.8 The trunc type-generic macro

1 The **trunc** functions use IEC 60559 rounding toward zero (regardless of the current rounding direction). The returned value is exact.

— **trunc**(± 0) returns ± 0.

— **trunc**$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is independent of the current rounding direction mode. The **trunc** type-generic macro may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments.

F.10.7 Remainder functions

F.10.7.1 The fmod type-generic macro

1 — **fmod**$(\pm 0, y)$ returns ± 0 for y not zero.

— **fmod**(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).

— **fmod**$(x, \pm\infty)$ returns x for x not infinite.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

3 The **double** version of **fmod** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double fmod(double x, double y)
{
      double result;
      result = remainder(fabs(x), (y = fabs(y)));
      if (signbit(result)) result += y;
      return copysign(result, x);
}
```

F.10.7.2 The remainder type-generic macro

1 — **remainder**$(\pm 0, y)$ returns ± 0 for y not zero.

— **remainder**(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).

— **remainder**$(x, \pm\infty)$ returns x for finite x.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.7.3 The remquo type-generic macro

1 The **remquo** functions follow the specifications for the **remainder** functions. They have no further specifications special to IEC 60559 implementations.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.8 Manipulation functions

F.10.8.1 The `copysign` type-generic macro

1 `copysign` is specified in the Appendix to IEC 60559.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.8.2 The `nan` functions

1 All IEC 60559 implementations support quiet NaNs, in all floating formats.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.8.3 The `nextafter` type-generic macro

1
- `nextafter`(x, y) raises the "overflow" and "inexact" floating-point exceptions for x finite and the function value infinite.
- `nextafter`(x, y) raises the "underflow" and "inexact" floating-point exceptions for the function value subnormal or zero and $x \neq y$.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.8.4 The `nexttoward` type-generic macro

1 No additional requirements beyond those on `nextafter`.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.9 Maximum, minimum, and positive difference functions

F.10.9.1 The `fdim` type-generic macro

1 No additional requirements.

F.10.9.2 The `fmax` type-generic macro

1 If just one argument is a NaN, the `fmax` functions return the other argument (if both arguments are NaNs, the functions return a NaN).

2 The returned value is exact and is independent of the current rounding direction mode.

3 The body of the `fmax` function might be[482)]

```
{ return (isgreaterequal(x, y) ||
      isnan(y)) ? x: y; }
```

F.10.9.3 The `fmin` type-generic macro

1 The `fmin` functions are analogous to the `fmax` functions (see F.10.9.2).

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.10 Floating multiply-add

F.10.10.1 The `fma` type-generic macro

1
- `fma`(x, y, z) computes $xy + z$, correctly rounded once.
- `fma`(x, y, z) returns a NaN and optionally raises the "invalid" floating-point exception if one of x and y is infinite, the other is zero, and z is a NaN.
- `fma`(x, y, z) returns a NaN and raises the "invalid" floating-point exception if one of x and y is infinite, the other is zero, and z is not a NaN.
- `fma`(x, y, z) returns a NaN and raises the "invalid" floating-point exception if x times y is an exact infinity and z is also an infinity but with the opposite sign.

482) Ideally, `fmax` would be sensitive to the sign of zero, for example `fmax`$(-0.0, +0.0)$ would return $+0$; however, implementation in software might be impractical.

F.10.11 Comparison macros

1 Relational operators and their corresponding comparison macros (7.12.14) produce equivalent result values, even if argument values are represented in wider formats. Thus, comparison macro arguments represented in formats wider than their semantic types are not converted to the semantic types, unless the wide evaluation method converts operands of relational operators to their semantic types. The standard wide evaluation methods characterized by **FLT_EVAL_METHOD** equal to 1 or 2 (5.2.4.2.2), do not convert operands of relational operators to their semantic types.

Annex G
(removed)
IEC 60559-compatible complex arithmetic

For C, this annex describes an extension introducing imaginary types. It has not found widespread support in the C community and has never been adapted to C++. Therefore these interfaces are not part of the C/C++ core.

Annex H
(informative)
Language independent arithmetic

H.1 Introduction

1 This annex documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1). LIA–1 is more general than IEC 60559 (Annex F) in that it covers integer and diverse floating-point arithmetics.

H.2 Types

1 The relevant C arithmetic types meet the requirements of LIA–1 types if an implementation adds notification of exceptional arithmetic operations and meets the 1 unit in the last place (ULP) accuracy requirement (LIA–1 subclause 5.2.8).

H.2.1 Boolean type

1 The LIA–1 data type Boolean is implemented by the C data type ~~bool~~**`bool`** with values of **`true`** and **`false`** ~~, all from .~~ .

H.2.2 Integer types

1 The signed C integer types **`int`**, **`long int`**, **`long long int`**, and the corresponding unsigned types are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values “integer_overflow” and “undefined”, then those types are LIA–1 conformant types. C’s unsigned integer types are “modulo” in the LIA–1 sense in that overflows or out-of-bounds results silently wrap. An implementation that defines signed integer types as also being modulo need not detect integer overflow, in which case, only integer divide-by-zero need be detected.

2 The parameters for the integer data types can be accessed by the following:

maxint **`INT_MAX`**, **`LONG_MAX`**, **`LLONG_MAX`**, **`UINT_MAX`**, **`ULONG_MAX`**, **`ULLONG_MAX`**

minint **`INT_MIN`**, **`LONG_MIN`**, **`LLONG_MIN`**

3 The parameter “bounded” is always true, and is not provided. The parameter “minint” is always 0 for the unsigned types, and is not provided for those types.

H.2.2.1 Integer operations

1 The integer operations on integer types are the following:

addI `x + y`

subI `x - y`

mulI ~~`x * y`~~ $x \times y$

divI, divtI `x / y`

remI, remtI `x % y`

negI `-x`

absI **`abs`**`(x)`, **`labs`**`(x)`, **`llabs`**`(x)`

eqI ~~`x == y`~~ $x \equiv y$

neqI ~~`x != y`~~ $x \neq y$

lssI `x < y`

leqI ~~`x <= y`~~ $x \leq y$

gtrI	`x > y`
geqI	~~x >= y~~ x ≥ y

where `x` and `y` are expressions of the same integer type.

H.2.3 Floating-point types

1 The C floating-point types **`float`**, **`double`**, and **`long double`** are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values "underflow", "floating_overflow", and "undefined", then those types are conformant with LIA–1. An implementation that uses IEC 60559 floating-point formats and operations (see Annex F) along with IEC 60559 status flags and traps has LIA–1 conformant types.

H.2.3.1 Floating-point parameters

1 The parameters for a floating-point data type can be accessed by the following:

r	**FLT_RADIX**
p	**FLT_MANT_DIG**, **DBL_MANT_DIG**, **LDBL_MANT_DIG**
emax	**FLT_MAX_EXP**, **DBL_MAX_EXP**, **LDBL_MAX_EXP**
emin	**FLT_MIN_EXP**, **DBL_MIN_EXP**, **LDBL_MIN_EXP**

2 The derived constants for the floating-point types are accessed by the following:

fmax	**FLT_MAX**, **DBL_MAX**, **LDBL_MAX**
fminN	**FLT_MIN**, **DBL_MIN**, **LDBL_MIN**
epsilon	**FLT_EPSILON**, **DBL_EPSILON**, **LDBL_EPSILON**
rnd_style	**FLT_ROUNDS**

H.2.3.2 Floating-point operations

1 The floating-point operations on floating-point types are the following:

addF	`x + y`
subF	`x - y`
mulF	~~x * y~~ x xy
divF	`x / y`
negF	`-x`
absF	`fabsf(x)`, `fabs(x)`, `fabsl(x)`
exponentF	`1.f+logbf(x)`, `1.0+logb(x)`, `1.L+logbl(x)`
scaleF	`scalbnf(x, n)`, `scalbn(x, n)`, `scalbnl(x, n)`,
	`scalblnf(x, li)`, `scalbln(x, li)`, `scalblnl(x, li)`
intpartF	`modff(x, &y)`, `modf(x, &y)`, `modfl(x, &y)`
fractpartF	`modff(x, &y)`, `modf(x, &y)`, `modfl(x, &y)`
eqF	~~x == y~~ x ≡ y
neqF	~~x != y~~ x ≠ y

lssF `x < y`

leqF ~~`x <= y`~~ `x` $\leq$ `y`

gtrF `x > y`

geqF ~~`x >= y`~~ `x` $\geq$ `y`

where `x` and `y` are expressions of the same floating-point type, `n` is of type **`int`**, and `li` is of type **`long int`**.

H.2.3.3 Rounding styles

1 This document requires all floating types to use the same radix and rounding style, so that only one identifier for each is provided to map to LIA–1.

2 The **`FLT_ROUNDS`** parameter can be used to indicate the LIA–1 rounding styles:

truncate ~~`FLT_ROUNDS == 0`~~ `FLT_ROUNDS` $\equiv$ `0`

nearest ~~`FLT_ROUNDS == 1`~~ `FLT_ROUNDS` $\equiv$ `1`

other ~~`FLT_ROUNDS != 0 && FLT_ROUNDS != 1`~~ `FLT_ROUNDS` $\neq$`0` $\wedge$`FLT_ROUNDS` $\neq$`1`

provided that an implementation extends **`FLT_ROUNDS`** to cover the rounding style used in all relevant LIA–1 operations, not just addition as in C.

H.2.4 Type conversions

1 The LIA–1 type conversions are the following type casts:

cvtI′ → I `(int)i, (long int)i, (long long int)i, (unsigned int)i, (unsigned long int)i, (unsigned long long int)i`

cvtF → I `(int)x, (long int)x, (long long int)x, (unsigned int)x, (unsigned long int)x, (unsigned long long int)x`

cvtI → F `(float)i, (double)i, (long double)i`

cvtF′ → F `(float)x, (double)x, (long double)x`

2 In the above conversions from floating to integer, the use of `(`*cast*`)x` can be replaced with `(`*cast*`)round(x)`, `(`*cast*`)rint(x)`, `(`*cast*`)nearbyint(x)`, `(`*cast*`)trunc(x)`, `(`*cast*`)ceil(x)`, or `(`*cast*`)floor(x)`. In addition, C's floating-point to integer conversion functions, **`lrint`**`()`, **`llrint`**`()`, **`lround`**`()`, and **`llround`**`()`, can be used. They all meet LIA–1's requirements on floating to integer rounding for in-range values. For out-of-range values, the conversions shall silently wrap for the modulo types.

3 The **`fmod`**`()` function is useful for doing silent wrapping to unsigned integer types, e.g., `fmod(fabs(rint(x)), 65536.0)` or ~~`(0.0 <= (y = fmod(rint(x), 65536.0))? y: 65536.0 + y)`~~ `(0.0` $\leq$`(y = fmod(rint(x), 65536.0))? y: 65536.0 + y)` will compute an integer value in the range 0.0 to 65535.0 which can then be converted to **`unsigned short int`**. But, the **`remainder`**`()` function is not useful for doing silent wrapping to signed integer types, e.g., `remainder(rint(x), 65536.0)` will compute an integer value in the range -32767.0 to $+32768.0$ which is not, in general, in the range of **`signed short int`**.

4 C's conversions (casts) from floating-point to floating-point can meet LIA–1 requirements if an implementation uses round-to-nearest (IEC 60559 default).

5 C's conversions (casts) from integer to floating-point can meet LIA–1 requirements if an implementation uses round-to-nearest.

H.3 Notification

1 Notification is the process by which a user or program is informed that an exceptional arithmetic operation has occurred. C's operations are compatible with LIA–1 in that C allows an implementation to cause a notification to occur when any arithmetic operation returns an exceptional value as defined in LIA–1 clause 5.

H.3.1 Notification alternatives

1 LIA–1 requires at least the following two alternatives for handling of notifications: setting indicators or trap-and-terminate. LIA–1 allows a third alternative: trap-and-resume.

2 An implementation need only support a given notification alternative for the entire program. An implementation may support the ability to switch between notification alternatives during execution, but is not required to do so. An implementation can provide separate selection for each kind of notification, but this is not required.

3 C allows an implementation to provide notification. C's **SIGFPE** (for traps) and **FE_INVALID**, **FE_DIVBYZERO**, **FE_OVERFLOW**, **FE_UNDERFLOW** (for indicators) can provide LIA–1 notification.

4 C's signal handlers are compatible with LIA–1. Default handling of **SIGFPE** can provide trap-and-terminate behavior, except for those LIA–1 operations implemented by math library function calls. User-provided signal handlers for **SIGFPE** allow for trap-and-resume behavior with the same constraint.

H.3.1.1 Indicators

1 C's <fenv.h> status flags are compatible with the LIA–1 indicators.

2 The following mapping is for floating-point types:

undefined	**FE_INVALID**, **FE_DIVBYZERO**
floating_overflow	**FE_OVERFLOW**
underflow	**FE_UNDERFLOW**

3 The floating-point indicator interrogation and manipulation operations are:

set_indicators	**feraiseexcept**(i)
clear_indicators	**feclearexcept**(i)
test_indicators	**fetestexcept**(i)
current_indicators	**fetestexcept**(**FE_ALL_EXCEPT**)

where i is an expression of type **int** representing a subset of the LIA–1 indicators.

4 C allows an implementation to provide the following LIA–1 required behavior: at program termination if any indicator is set the implementation shall send an unambiguous and "hard to ignore" message (see LIA–1 subclause 6.1.2)

5 LIA–1 does not make the distinction between floating-point and integer for "undefined". This documentation makes that distinction because <fenv.h> covers only the floating-point indicators.

H.3.1.2 Traps

1 C is compatible with LIA–1's trap requirements for arithmetic operations, but not for math library functions (which are not permitted to invoke a user's signal handler for **SIGFPE**). An implementation can provide an alternative of notification through termination with a "hard-to-ignore" message (see LIA–1 subclause 6.1.3).

2 LIA–1 does not require that traps be precise.

3 C does require that **SIGFPE** be the signal corresponding to LIA–1 arithmetic exceptions, if there is any signal raised for them.

4 C supports signal handlers for **`SIGFPE`** and allows trapping of LIA–1 arithmetic exceptions. When LIA–1 arithmetic exceptions do trap, C's signal-handler mechanism allows trap-and-terminate (either default implementation behavior or user replacement for it) or trap-and-resume, at the programmer's option.

Annex I
(informative)
Common warnings

1 An implementation may generate warnings in many situations, none of which are specified as part of this document. The following are a few of the more common situations.

2
- A new `struct` or `union` type appears in a function prototype (6.2.1, 6.7.2.3).
- A block with initialization of an object that has automatic storage duration is jumped into (6.2.4).
- An implicit narrowing conversion is encountered, such as the assignment of a `long int` or a `double` to an `int`, or a pointer to `void` to a pointer to any type other than a character type (6.3).
- A hexadecimal floating constant cannot be represented exactly in its evaluation format (6.4.4.2).
- An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (6.4.4.4).
- The characters `/*` are found in a comment (6.4.7).
- An "unordered" binary operator (not comma, ~~&&, or ||~~ ∧, or ∨) contains a side effect to an lvalue in one operand, and a side effect to, or an access to the value of, the identical lvalue in the other operand (6.5).
- A function is called but no prototype has been supplied (6.5.2.2).
- An object is defined but not used (6.7).
- A value is given to an object of an enumerated type other than by assignment of an enumeration constant that is a member of that type, or an enumeration object that has the same type, or the value of a function that returns the same enumerated type (6.7.2.2).
- An aggregate has a partly bracketed initialization (6.7.10).
- A statement cannot be reached (6.8).
- A statement with no apparent effect is encountered (6.8).
- A constant expression is used as the controlling expression of a selection statement (6.8.4).
- An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (6.10.1).
- An unrecognized `#pragma` directive is encountered (6.10.6).

Annex J
(informative)
Portability issues

1 This annex collects some information about portability that appears in this document.

J.1 Unspecified behavior

1 The following are unspecified:

- The manner and timing of static initialization (5.1.2).
- The termination status returned to the hosted environment if the return type of **`main`** is not compatible with int (5.1.2.2.3).
- The values of objects that are neither lock-free atomic objects nor of type **`volatile`** **`sig_atomic_t`** and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).
- The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).
- The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).
- The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).
- The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).
- How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).
- Many aspects of the representations of types (6.2.6).
- The relative order of any two storage instances in the address space (6.2.6.1).
- The value of padding bytes when storing values in structures or unions (6.2.6.1).
- The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).
- The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).
- The values of any padding bits in integer representations (6.2.6.2).
- Whether two string literals result in distinct arrays (6.4.5).
- The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, ~~&&, ||~~ $\wedge$, $\vee$, `?:`, and comma operators (6.5).
- The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).
- The order of side effects among compound literal initialization list expressions (6.5.2.5).
- The order in which the operands of an assignment operator are evaluated (6.5.17).
- The alignment of ~~the addressable storage unit allocated to hold a bit-field~~ a pack (6.7.2.1).
- Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.5).

— Whether or not a size expression is evaluated when it is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator (6.7.8.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.12).

— ~~The layout of storage for function parameters (6.9.1).~~ When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `(`, and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `(`, whether that is considered a nested replacement (6.10.3).

— The order in which **#** and **##** operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— The line number of a preprocessing token, in particular **__LINE__**, that spans multiple physical lines (6.10.4).

— The line number of a preprocessing directive that spans multiple physical lines (6.10.4).

— The line number of a macro invocation that spans multiple physical or logical lines (6.10.4).

— The line number following a directive of the form **#line __LINE__** *new-line* (6.10.4).

— The state of the floating-point status flags when execution passes from a part of the program translated with **FENV_ACCESS** "off" to a part translated with **FENV_ACCESS** "on" (7.6.1).

— The order in which **feraiseexcept** raises floating-point exceptions, except as stated in F.8.6 (7.6.2.3).

— Whether **math_errhandling** is a macro or an identifier with external linkage (7.12).

— The results of the **frexp** functions when the specified value is not a floating-point number (7.12.6.4).

— The numeric result of the **ilogb** ~~functions~~ type-generic macro when the correct value is outside the range of the return type (~~??, ??~~7.12.6.5, F.10.3.5).

— The result of rounding when the value is out of range (~~??, ??, ??~~7.12.9.5, 7.12.9.7, F.10.6.5).

— The value stored by the **remquo** functions in the object pointed to by `quo` when `y` is zero (7.12.10.3).

— Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.14).

— Whether **setjmp** is a macro or an identifier with external linkage (7.13).

— Whether **va_copy** and **va_end** are macros or identifiers with external linkage (7.16.1).

— The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an `a` or `A` conversion specifier (7.21.6.1, 7.29.2.1).

— The value of the file position indicator after a successful call to the **ungetc** function for a text stream, or the **ungetwc** function for any stream, until all pushed-back characters are read or discarded (7.21.7.10, 7.29.3.10).

— The details of the value stored by the **fgetpos** function (7.21.9.1).

— The details of the value returned by the **ftell** function for a text stream (7.21.9.4).

— Whether the **`strtod`**, **`strtof`**, ~~**`strtold`**, **`wcstod`**, **`wcstof`**, and **`wcstold`** functions~~ and **`strtold`** type-generic macros convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (~~??, ??). The order and contiguity of storage allocated by successive calls to the **`calloc`**, **`malloc`**, **`realloc`**, and **`aligned_alloc`** functions (??~~7.22.1.3).

— ~~The amount of storage allocated by a successful~~ If a call to the **`calloc`**, **`malloc`**, **`realloc`**, or **`aligned_alloc`** function ~~when~~ requesting 0 bytes ~~was requested (??~~fails or returns a storage instance of size zero (7.22.4).

— Whether a call to the **`atexit`** function that does not happen before the **`exit`** function is called will succeed (7.22.5.2).

— Whether a call to the **`at_quick_exit`** function that does not happen before the **`quick_exit`** function is called will succeed (7.22.5.3).

— Which of two elements that compare as equal is matched by the **`bsearch`** ~~function (??~~type-generic macro (7.22.6.1).

— The order of two elements that compare as equal in an array sorted by the **`qsort`** function (7.22.6.2).

— The order in which destructors are invoked by **`thrd_exit`** (7.26.5.5).

— Whether calling **`tss_delete`** on a key while another thread is executing destructors affects the number of invocations of the destructors associated with the key on that thread (7.26.6.2).

— The encoding of the calendar time returned by the **`time`** function (7.27.2.4).

— The characters stored by the **`strftime`** or **`wcsftime`** function if any of the time values being converted is outside the normal range (7.27.3.5, 7.29.5.1).

— Whether an encoding error occurs if a **`wchar_t`** value that does not correspond to a member of the extended character set appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by **`wcrtomb`** (7.29.1).

— The conversion state after an encoding error occurs (7.29.6.3.2, 7.29.6.3.3, 7.29.6.4.1, 7.29.6.4.2,

— The resulting value when the "invalid" floating-point exception is raised during IEC 60559 floating to integer conversion (F.4).

— Whether conversion of non-integer IEC 60559 floating values to integer raises the "inexact" floating-point exception (F.4).

— Whether or when library functions in `<math.h>` raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.10).

— Whether or when library functions in `<math.h>` raise an undeserved "underflow" floating-point exception in an IEC 60559 conformant implementation (F.10).

— The exponent value stored by **`frexp`** for a NaN or infinity (F.10.3.4).

— The numeric result returned by the **`lrint`**, **`llrint`**, **`lround`**, and **`llround`** ~~functions~~ type-generic macros if the rounded value is outside the range of the return type (~~??, ??). The sign of one part of the **`complex`** result of several math functions for certain special cases in IEC 60559 compatible implementations (??, ??, ??, ??, ??, ??, ??, ??~~F.10.6.5, F.10.6.7).

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

- A "shall" or "shall not" requirement that appears outside of a constraint is violated (Clause 4).
- A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).
- Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).
- A program in a hosted environment does not define a function named **`main`** using one of the specified forms (5.1.2.2.1).
- The execution of a program contains a data race (5.1.2.4).
- A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).
- An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.1).
- The same identifier has both internal and external linkage in the same translation unit (6.2.2).
- An object is referred to outside of its lifetime (6.2.4).
- The value of a pointer to an object whose lifetime has ended is used (6.2.4).
- The value of an object with automatic storage duration is used while it is indeterminate (6.2.4 , 6.7.12, 6.8).
- A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).
- A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).
- Two declarations of the same object or function specify types that are not compatible (6.2.7).
- A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).
- Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).
- Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).
- An lvalue does not designate an object when evaluated (6.3.2.1).
- A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).
- An ~~lvalue designating an object of automatic storage duration that could have been declared with the **`register`** storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1). An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1). An~~ attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **`void`**) is applied to a void expression (6.3.2.2).
- Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).
- Conversion of **`nullptr`** to a type that is not a pointer type (6.3.2.3).

— Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

— A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

— An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).

— A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

— A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

— The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— The identifier `__func__` is explicitly declared (6.4.2.2).

— The program attempts to modify a string literal (6.4.5).

— The characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).

— A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

— An exceptional condition occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

— For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after default argument promotion are not compatible with the types of the parameters (6.5.2.2).

— A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

— A member of an atomic structure or union is accessed (6.5.2.3).

— The operand of the unary `*` operator has an invalid value (6.5.3.2).

— A pointer is converted to other than an integer or pointer type (6.5.4).

— The value of the second operand of the `/` or `%` operator is zero (6.5.5).

— If the quotient `a/b` is not representable, the behavior of both `a/b` and `a%b` (6.5.5).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).

— Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.5.6).

— The result of subtracting two pointers is not representable in an object of type **ptrdiff_t** (6.5.6).

— An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

— An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type (6.5.7).

— Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.9).

— Either of the second or third operands of a conditional operator is **nullptr**, and the other is an integer constant expression of value 0 (6.5.16).

— An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.17.1).

— An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, ~~**_Alignof**~~ **alignof** expressions, or immediately-cast floating constants; or contains casts (outside operands to **sizeof** and ~~**_Alignof**~~ **alignof** operators) other than conversions of arithmetic types to integer types (6.6).

— A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

— An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, or ~~**_Alignof**~~ **alignof** expressions; or contains casts (outside operands to **sizeof** or ~~**_Alignof**~~ **alignof** operators) other than conversions of arithmetic types to arithmetic types (6.6).

— The value of an object is accessed by an array-subscript [], member-access . or ~~->~~ →, address &, or indirection * operator or a pointer cast in creating an address constant (6.6).

— An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

— A function is declared at block scope with an explicit storage-class specifier other than **extern** (6.7.1).

— A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

— An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

— When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

— An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

— The specification of a function type includes any type qualifiers (6.7.3).

— Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

— An object which has been modified is accessed through a ~~restrict-qualified~~ noalias attributed pointer to a const-qualified type, or through a ~~restrict-qualified~~ noalias attributed pointer and another pointer that are not both based on the same object (6.7.15.3.1).

— A ~~restrict-qualified~~ noalias attributed pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.15.3.1).

— A function with external linkage is declared with an **inline** function specifier, but is not also defined in the same translation unit (6.7.5).

— A function declared with a **_Noreturn** function specifier returns to its caller (**??**).

— The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.6).

— Declarations of an object in different translation units have different alignment specifiers (6.7.6).

— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.8.1).

— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.8.2).

— In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.8.2).

— A declaration of an array parameter includes the keyword **static** within the [and] and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.8.3).

— A storage-class specifier or type qualifier modifies the keyword **void** as a function parameter type list (6.7.8.3).

— In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list) (6.7.8.3).

— The value of an unnamed member of a structure or union is used (6.7.12).

— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.12).

— The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.12).

— The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.12).

— An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

— An adjusted parameter type in a function definition is not a complete object type (6.9.1).

— A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

— The } that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

— An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

— A non-directive preprocessing directive is executed (6.10).

— The token **defined** is generated during the expansion of a **#if** or **#elif** preprocessing directive, or the use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

— The **#include** preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

— The character sequence in an **#include** preprocessing directive does not start with a letter (6.10.2).

— There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).

— The result of the preprocessing operator **#** is not a valid character string literal (6.10.3.2).

— The result of the preprocessing operator **##** is not a valid preprocessing token (6.10.3.3).

— The **#line** preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.4).

— A non-**STDC #pragma** preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).

— A **#pragma STDC** preprocessing directive does not match one of the well-defined forms (6.10.6).

— The name of a predefined macro, or the identifier **defined**, is the subject of a **#define** or **#undef** preprocessing directive (6.10.8).

— An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., **memmove**) (Clause 7).

— A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

— A header is included within an external declaration or definition (7.1.2).

— A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

— A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

— The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

— The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

— The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).

— An argument to a library function has an invalid value or a type not expected by a function with a variable number of arguments (7.1.4).

— The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

— The macro definition of **assert** is suppressed in order to access an actual function (7.2).

— The argument to the **assert** macro does not have a scalar type (7.2).

— The `CX_LIMITED_RANGE`, `FENV_ACCESS`, or `FP_CONTRACT` pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

— The value of an argument to a character handling function is neither equal to the value of `EOF` nor representable as an `unsigned char` (7.4).

— A macro definition of `errno` is suppressed in order to access an actual object, or the program defines an identifier with the name `errno` (7.5).

— Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma "off" (7.6.1).

— The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.2).

— The `fesetexceptflag` function is used to set floating-point status flags that were not specified in the call to the `fegetexceptflag` function that provided the value of the corresponding `fexcept_t` object (7.6.2.4).

— The argument to `fesetenv` or `feupdateenv` is neither an object set by a call to `fegetenv` or `feholdexcept`, nor is it an environment macro (7.6.4.3, 7.6.4.4).

— The value of the result of an integer arithmetic or conversion function cannot be represented (~~??, ??, ??, ??, 7.22.7.1, ??~~7.22.7.2, 7.22.1).

— The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).

— The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).

— A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).

— An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.14).

— A macro definition of `setjmp` is suppressed in order to access an actual function, or the program defines an external identifier with the name `setjmp` (7.13).

— An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).

— The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).

— After a `longjmp`, there is an attempt to access the value of an object of automatic storage duration that does not have volatile-qualified type, local to the function containing the invocation of the corresponding `setjmp` macro, that was changed between the `setjmp` invocation and `longjmp` call (7.13.2.1).

— The program specifies an invalid pointer to a signal handler function (7.14.1.1).

— A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

— A signal handler called in response to `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

— A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).

— A signal occurs other than as the result of calling the **abort** or **raise** function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as **volatile sig_atomic_t**, or calls any function in the standard library other than the **abort** function, the **_Exit** function, the **quick_exit** function, the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free, the **atomic_is_lock_free** function with any atomic argument, or the **signal** function (for the same signal number) (7.14.1.1).

— The value of **errno** is referred to after a signal occurred other than as the result of calling the **abort** or **raise** function and the corresponding signal handler obtained a **SIG_ERR** return from a call to the **signal** function (7.14.1.1).

— A signal is generated by an asynchronous signal handler (7.14.1.1).

— The **signal** function is used in a multi-threaded program (7.14.1.1).

— A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized **va_list** object, or before the **va_start** macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

— The macro **va_arg** is invoked using the parameter `ap` that was passed to a function that invoked the macro **va_arg** with the same parameter (7.16).

— A macro definition of **va_start**, **va_arg**, **va_copy**, or **va_end** is suppressed in order to access an actual function, or the program defines an external identifier with the name **va_copy** or **va_end** (7.16.1).

— The **va_start** or **va_copy** macro is invoked without a corresponding invocation of the **va_end** macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

— The type parameter to the **va_arg** macro is not such that a pointer to an object of that type can be obtained simply by postfixing a * (7.16.1.1).

— The **va_arg** macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

— The **va_copy** or **va_start** macro is called to initialize a **va_list** that was previously initialized by either macro without an intervening invocation of the **va_end** macro for the same **va_list** (7.16.1.2, 7.16.1.4).

— The parameter *parmN* of a **va_start** macro is declared with ~~the **register** storage class, with~~ a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.16.1.4).

— The macro definition of a generic function is suppressed in order to access an actual function (7.17.1).

— The *type* parameter of an **offsetof** macro defines a new type (6.10.8.1).

— The *member-designator* parameter of an **offsetof** macro is an invalid right operand of the . operator for the *type* parameter ~~, or designates a bit-field (7.19~~(6.10.8.1).

— The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.20.4).

— A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.21.2).

— Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).

— The value of a pointer to a `FILE` object is used after the associated file is closed (7.21.3).

— The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).

— The string pointed to by the `mode` argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.21.5.3).

— An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).

— An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.21.5.6).

— There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).

— An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%%` (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.21.6.1, 7.29.2.1).

— The number of input items assigned by a formatted input function is greater than `INT_MAX` (7.21.6.2, 7.29.2.2).

— The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.21.6.2, 7.29.2.2).

— A `c`, `s`, or `[` conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is `s` or `[`) (7.21.6.2, 7.29.2.2).

— A `c`, `s`, or `[` conversion specifier with an `l` qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

— The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

— The **`vfprintf`**, **`vfscanf`**, **`vprintf`**, **`vscanf`**, **`vsnprintf`**, **`vsprintf`**, **`vsscanf`**, **`vfwprintf`**, **`vfwscanf`**, **`vswprintf`**, **`vswscanf`**, **`vwprintf`**, or **`vwscanf`** function is called with an improperly initialized **`va_list`** argument, or the argument is used (other than in an invocation of **`va_end`**) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

— The contents of the array supplied in a call to the **`fgets`** or **`fgetws`** function are used after a read error occurred (7.21.7.2, 7.29.3.2).

— The file position indicator for a binary stream is used after a call to the **`ungetc`** function where its value was zero before the call (7.21.7.10).

— The file position indicator for a stream is used after an error occurred during a call to the **`fread`** or **`fwrite`** function (7.21.8.1, 7.21.8.2).

— A partial element read by a call to the **`fread`** function is used (7.21.8.1).

— The **`fseek`** function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the **`ftell`** function on a stream associated with the same file or `whence` is not **`SEEK_SET`** (7.21.9.2).

— The **`fsetpos`** function is called to set a position that was not returned by a previous successful call to the **`fgetpos`** function on a stream associated with the same file (7.21.9.3).

— A non-null pointer returned by a call to the **`calloc`**, **`malloc`**, **`realloc`**, or **`aligned_alloc`** function with a zero requested size is used to access an object (7.22.4).

— The value of a pointer that refers to ~~space~~ a storage instance deallocated by a call to the **`free`** or **`realloc`** function is used (7.22.4).

— The pointer argument to the **`free`** or **`realloc`** function does not match a pointer earlier returned by a ~~memory~~ storage management function, or the ~~space~~ storage instance has been deallocated by a call to **`free`** or **`realloc`** (7.22.4.3, 7.22.4.5).

— The value of the object allocated by the **`malloc`** function is used (7.22.4.4).

— The values of any bytes in a new object allocated by the **`realloc`** function beyond the size of the old object are used (7.22.4.5).

— The program calls the **`exit`** or **`quick_exit`** function more than once, or calls both functions (7.22.5.4, 7.22.5.7).

— During the call to a function registered with the **`atexit`** or **`at_quick_exit`** function, a call is made to the **`longjmp`** function that would terminate the call to the registered function (7.22.5.4, 7.22.5.7).

— The string set up by the **`getenv`** or **`strerror`** function is modified by the program (7.22.5.6, 7.24.6.3).

— A signal is raised while the **`quick_exit`** function is executing (7.22.5.7).

— A command is executed through the **`system`** function in a way that is documented as causing termination or some other form of undefined behavior (7.22.5.8).

— A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.6).

— The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.22.6).

— The array being searched by the **bsearch** ~~function~~ type-generic macro does not have its elements in proper order (7.22.6.1).

— The current conversion state is used by a multibyte/wide character conversion function after changing the **LC_CTYPE** category (7.22.8).

— A string or wide string utility function is instructed to access an array beyond the end of an object (7.24.1.1~~, ??~~).

— A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.24.1.1~~, ??~~).

— The contents of the destination array are used after a call to the **strxfrm**, **strftime**, **wcsxfrm**, or **wcsftime** function in which the specified length was too small to hold the entire null-terminated result (~~??, 7.27.3.5, ??, 7.29.5.1~~7.24.4.5, 7.27.3.5).

— The first argument in the very first call to the **strtok** or **wcstok** is a null pointer (~~??, ??~~7.24.5.8).

— The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.25).

— A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.25).

— A non-recursive mutex passed to **mtx_lock** is locked by the calling thread (7.26.4.3).

— The mutex passed to **mtx_timedlock** does not support timeout (7.26.4.4).

— The mutex passed to **mtx_unlock** is not locked by the calling thread (7.26.4.6).

— The thread passed to **thrd_detach** or **thrd_join** was previously detached or joined with another thread (7.26.5.3, 7.26.5.6).

— The **tss_create** function is called from within a destructor (7.26.6.1).

— The key passed to **tss_delete**, **tss_get**, or **tss_set** was not returned by a call to **tss_create** before the thread commenced executing destructors (7.26.6.2, 7.26.6.3, 7.26.6.4).

— At least one member of the broken-down time passed to **asctime** contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1).

— The argument corresponding to an `s` specifier without an `l` qualifier in a call to the **fwprintf** function does not point to a valid multibyte character sequence that begins in the initial shift state (7.29.2.11).

— ~~In a call to the **wcstok** function, the object pointed to by `ptr` does not have the value stored by the previous call for the same wide string (??).~~ An **mbstate_t** object is used inappropriately (7.29.6).

— The value of an argument of type **wint_t** to a wide character classification or case mapping function is neither equal to the value of **WEOF** nor representable as a **wchar_t** (7.30.1).

— The **iswctype** function is called using a different **LC_CTYPE** category from the one in effect for the call to the **wctype** function that returned the description (7.30.2.2.1).

— The **towctrans** function is called using a different **LC_CTYPE** category from the one in effect for the call to the **wctrans** function that returned the description (7.30.3.2.1).

J.3 Implementation-defined behavior

1 A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

J.3.1 Translation

1 — How a diagnostic is identified (3.10, 5.1.1.3).

— Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

J.3.2 Environment

1 — The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

— The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

— The effect of program termination in a freestanding environment (5.1.2.1).

— An alternative manner in which the `main` function may be defined (5.1.2.2.1).

— The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

— What constitutes an interactive device (5.1.2.3).

— Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

— The set of signals, their semantics, and their default handling (7.14).

— Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

— Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

— The set of environment names and the method for altering the environment list used by the `getenv` function (7.22.5.6).

— The manner of execution of the string by the `system` function (7.22.5.8).

J.3.3 Identifiers

1 — Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

— The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

J.3.4 Characters

1 — The number of bits in a byte (3.6).

— The values of the members of the execution character set (5.2.1).

— The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

— The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

— Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

— The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

— The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

— The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

— The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

— Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

— The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

— The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5). ~~The encoding of any of **wchar_t**, **char16_t**, and **char32_t** where the corresponding standard encoding macro (**__STDC_ISO_10646__**, **__STDC_UTF_16__**, or **__STDC_UTF_32__**) is not defined (??).~~

J.3.5 Integers

1 — Any extended integer types that exist in the implementation (6.2.5).

— The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

— The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

— The results of some bitwise operations on signed integers (6.5).

J.3.6 Floating point

1 — The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

— The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

— The rounding behaviors characterized by non-standard values of **FLT_ROUNDS** (5.2.4.2.2).

— The evaluation methods characterized by non-standard negative values of **FLT_EVAL_METHOD** (5.2.4.2.2).

— The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

— The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

— How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

— Whether and how floating expressions are contracted when not disallowed by the **FP_CONTRACT** pragma (6.5).

— The default state for the `FENV_ACCESS` pragma (7.6.1).

— Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

— The default state for the `FP_CONTRACT` pragma (7.12.2).

J.3.7 Arrays and pointers

1 — The result of converting a pointer to an integer or vice versa (6.3.2.3).

— The size of the result of subtracting two pointers to elements of the same array (6.5.6).

J.3.8 Hints

1 — The extent to which suggestions made by using the ~~`register` storage-class specifier are effective (6.7.1). The extent to which suggestions made by using the~~ `inline` function specifier are effective (6.7.5).

J.3.9 Structures, unions, and enumerations

1 — Whether a ~~"plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.7.2, 6.7.2.1). Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1). Whether atomic types are permitted for bit-fields (6.7.2.1). Whether a bit-field~~ `core::alias` member can straddle a storage-unit boundary (6.7.2.1).

— The order ~~of allocation of bit-fields within a unit~~ and representation of `core::alias` member within a pack (6.7.2.1).

— The alignment of ~~non-bit-field~~ `core::noalias` members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

— The integer type compatible with each enumerated type (6.7.2.2).

J.3.10 Qualifiers

1 — What constitutes an access to an object that has volatile-qualified type (6.7.3).

J.3.11 Preprocessing directives

1 — The locations within `#pragma` directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

— How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

— Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

— Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

— The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

— How the named source file is searched for in an included `" "` delimited header (6.10.2).

— The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

— The nesting limit for `#include` processing (6.10.2).

— Whether the **#** operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

— The behavior on each recognized non-`STDC` `#pragma` directive (6.10.6).

— The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8.1).

J.3.12 Library functions

1 — Any library facilities available to a freestanding program, other than the minimal set required by Clause 4 (5.1.2.1).

— The format of the diagnostic printed by the `assert` macro (7.2.1.1).

— The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).

— Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

— Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

— The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

— Domain errors for the mathematics functions, other than those required by this document (7.12.1).

— The values returned by the mathematics functions on domain errors or pole errors (7.12.1).

— The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

— Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

— Whether a domain error occurs or zero is returned when a `remainder` function has a second argument of zero (7.12.10.2).

— The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

— Whether a domain error occurs or zero is returned when a `remquo` function has a second argument of zero (7.12.10.3).

— Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

— The null pointer constant to which the macro `NULL` expands (6.10.8.1).

— Whether the last line of a text stream requires a terminating new-line character (7.21.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

— The number of null characters that may be appended to data written to a binary stream (7.21.2).

— Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).
- The characteristics of file buffering (7.21.3).
- Whether a zero-length file actually exists (7.21.3).
- The rules for composing valid file names (7.21.3).
- Whether the same file can be simultaneously open multiple times (7.21.3).
- The nature and choice of encodings used for multibyte characters in files (7.21.3).
- The effect of the `remove` function on an open file (7.21.4.1).
- The effect if a file with the new name exists prior to a call to the `rename` function (7.21.4.2).
- Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).
- Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).
- The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).
- The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).
- The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).
- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).
- The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).
- The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, ~~`strtold`, `wcstod`, `wcstof`, or `wcstold` function (??, ??~~or `strtold`, type-generic macro (7.22.1.3).
- Whether or not the `strtod`, `strtof`, ~~`strtold`, `wcstod`, `wcstof`, or `wcstold` function~~ or `strtold`, type-generic macro sets `errno` to `ERANGE` when underflow occurs (~~??, ??~~7.22.1.3).
- Whether the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions return a null pointer or a pointer to ~~an allocated object~~ a storage instance when the size requested is zero (7.22.4).
- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.22.5.1, 7.22.5.5).
- The termination status returned to the host environment by the `abort`, `exit`, `_Exit`, or `quick_exit` function (7.22.5.1, 7.22.5.4, 7.22.5.5, 7.22.5.7).
- The value returned by the `system` function when its argument is not a null pointer (7.22.5.8).
- The range and precision of times representable in `clock_t` and `time_t` (7.27).
- The local time zone and Daylight Saving Time (7.27.1).
- The era for the `clock` function (7.27.2.1).
- The `TIME_UTC` epoch (7.27.2.5).
- The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the `"C"` locale (7.27.3.5, 7.29.5.1).
- Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

J.3.13 Architecture

1 — The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.20).

— The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

— The number, order, and encoding of bytes in any object (when not explicitly specified in this document) (6.2.6.1).

— Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

— Valid alignment values other than those returned by an ~~`_Alignof`~~ `alignof` expression for fundamental types, if any (6.2.8).

— The value of the result of the **`sizeof`** and ~~`_Alignof`~~ `alignof` operators (6.5.3.4).

J.4 Locale-specific behavior

1 The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

— Additional members of the source and execution character sets beyond the basic character set (5.2.1).

— The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.1).

— The shift states used for the encoding of multibyte characters (5.2.1.1).

— The direction of writing of successive printing characters (5.2.2).

— The decimal-point character (7.1.1).

— The set of printing characters (7.4, 7.30.2).

— The set of control characters (7.4, 7.30.2).

— The sets of characters tested for by the **`isalpha`**, **`isblank`**, **`islower`**, **`ispunct`**, **`isspace`**, **`isupper`**, **`iswalpha`**, **`iswblank`**, **`iswlower`**, **`iswpunct`**, **`iswspace`**, or **`iswupper`** functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).

— The native environment (7.11.1.1).

— Additional subject sequences accepted by the numeric conversion functions (7.22.1~~, ??~~).

— The collation sequence of the execution character set (~~??, ??~~7.24.4.3).

— The contents of the error message strings set up by the **`strerror`** function (7.24.6.3).

— The formats for time and date (7.27.3.5, 7.29.5.1).

— Character mappings that are supported by the **`towctrans`** function (7.30.1).

— Character classifications that are supported by the **`iswctype`** function (7.30.1).

J.5 Common extensions

1 The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, extra library functions declared in standard headers, or predefined macros with names that do not begin with an underscore.

J.5.1 Environment arguments

1 In a hosted environment, the **main** function receives a third argument, **char** *envp[], that points to a null-terminated array of pointers to **char**, each of which points to a string that provides information about the environment for this execution of the program (5.1.2.2.1).

J.5.2 Specialized identifiers

1 Characters other than the underscore _, letters, and digits, that are not part of the basic source character set (such as the dollar sign $, or characters in national character sets) may appear in an identifier (6.4.2).

J.5.3 Lengths and cases of identifiers

1 All characters in identifiers (with or without external linkage) are significant (6.4.2).

J.5.4 Scopes of identifiers

1 A function identifier, or the identifier of an object the declaration of which contains the keyword **extern**, has file scope (6.2.1).

J.5.5 Writable string literals

1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

J.5.6 Other arithmetic types

1 Additional arithmetic types, such as __int128 or **double double**, and their appropriate conversions are defined (6.2.5, 6.3.1). Additional floating types may have more range or precision than **long double**, may be used for evaluating expressions of other floating types, and may be used to define **float_t** or **double_t**. Additional floating types may also have less range or precision than **float**.

J.5.7 Function pointer casts

1 A pointer to an object or to **void** may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4).

2 A pointer to a function may be cast to a pointer to an object or to **void**, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).

~~A bit-field may be declared with a type other than **_Bool**, **unsigned int**, or **signed int**, with an appropriate maximum width (6.7.2.1).~~

J.5.8 The fortran keyword

1 The fortran function specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that a different representation for the external name is to be generated~~(??)~~.

J.5.9 The asm keyword

1 The asm keyword may be used to insert assembly language directly into the translator output (6.8). The most common implementation is via a statement of the form:

```
asm (character-string-literal);
```

J.5.10 Multiple external definitions

1 There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern**; if the definitions disagree, or more than one is initialized, the behavior is undefined (6.9.2).

J.5.11 Predefined macro names

1 Macro names that do not begin with an underscore, describing the translation and execution environments, are defined by the implementation before translation begins (6.10.8).

J.5.12 Floating-point status flags

1 If any floating-point status flags are set on normal termination after all calls to functions registered by the `atexit` function have been made (see 7.22.5.4), the implementation writes some diagnostics indicating the fact to the `stderr` stream, if it is still open,

J.5.13 Extra arguments for signal handlers

1 Handlers for specific signals are called with extra arguments in addition to the signal number (7.14.1.1).

J.5.14 Additional stream types and file-opening modes

1 Additional mappings from files to streams are supported (7.21.2).

2 Additional file-opening modes may be specified by characters appended to the `mode` argument of the `fopen` function (7.21.5.3).

J.5.15 Defined file position indicator

1 The file position indicator is decremented by each successful call to the `ungetc` or `ungetwc` function for a text stream, except if its value was zero before a call (7.21.7.10, 7.29.3.10).

J.5.16 Math error reporting

1 Functions declared in `<complex.h>` and `<math.h>` raise `SIGFPE` to report errors instead of, or in addition to, setting `errno` or raising floating-point exceptions (7.3, 7.12).

J.6 Reserved identifiers and keywords

1 A lot of identifier preprocessing tokens are used for specific purposes in regular clauses or appendices from translation phase 3 onwards. Using any of these for a purpose different from their description in this document, even if the use is in a context where they are normatively permitted, may have an impact on the portability of code and should thus be avoided.

J.6.1 Rule based identifiers

1 The following 27 regular expressions characterize identifiers that are systematically reserved by some clause this document.

```
atomic_[a-z][a-zA-Z0-9_]*
ATOMIC_[A-Z][a-zA-Z0-9_]*
_[a-zA-Z_][a-zA-Z0-9_]*
cnd_[a-z][a-zA-Z0-9_]*
E[0-9A-Z][a-zA-Z0-9_]*
FE_[A-Z][a-zA-Z0-9_]*
INT[a-zA-Z0-9_]*_C
INT[a-zA-Z0-9_]*_MAX
INT[a-zA-Z0-9_]*_MIN
int[a-zA-Z0-9_]*_t
is[a-z][a-zA-Z0-9_]*
LC_[A-Z][a-zA-Z0-9_]*
mem[a-z][a-zA-Z0-9_]*
mtx_[a-z][a-zA-Z0-9_]*
PRI[a-zX][a-zA-Z0-9_]*
SCN[a-zX][a-zA-Z0-9_]*
SIG[A-Z][a-zA-Z0-9_]*
SIG_[A-Z][a-zA-Z0-9_]*
str[a-z][a-zA-Z0-9_]*
thrd_[a-z][a-zA-Z0-9_]*
TIME_[A-Z][a-zA-Z0-9_]*
to[a-z][a-zA-Z0-9_]*
tss_[a-z][a-zA-Z0-9_]*
UINT[a-zA-Z0-9_]*_C
UINT[a-zA-Z0-9_]*_MAX
uint[a-zA-Z0-9_]*_t
wcs[a-z][a-zA-Z0-9_]*
```

2 The following ~~447~~ 469 identifiers or keywords match these patterns and have particular semantics provided by this document.

For features that are obsolescent, see below.

~~_Alignas~~
~~__alignas_is_defined~~
~~_Alignof~~
~~__alignof_is_defined~~ __address_independent__
~~_Atomic~~ __alias__
atomic_bool
ATOMIC_BOOL_LOCK_FREE
atomic_char
atomic_char16_t
ATOMIC_CHAR16_T_LOCK_FREE
atomic_char32_t
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit
atomic_exchange
atomic_exchange_explicit
atomic_fetch_
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag
atomic_flag_clear
atomic_flag_clear_explicit
~~ATOMIC_FLAG_INIT~~
atomic_flag_test_and_set
atomic_flag_test_and_set_explicit
atomic_init
atomic_int
atomic_int_fast16_t
atomic_int_fast32_t
atomic_int_fast64_t
atomic_int_fast8_t
atomic_int_least16_t
atomic_int_least32_t
atomic_int_least64_t
atomic_int_least8_t
ATOMIC_INT_LOCK_FREE
atomic_intmax_t
atomic_intptr_t
atomic_is_lock_free
atomic_llong
ATOMIC_LLONG_LOCK_FREE
atomic_load
atomic_load_explicit
atomic_long
ATOMIC_LONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
atomic_ptrdiff_t
atomic_schar
atomic_short
ATOMIC_SHORT_LOCK_FREE
atomic_signal_fence
atomic_size_t
atomic_store
atomic_store_explicit
atomic_thread_fence
atomic_type
atomic_uchar
atomic_uint
atomic_uint_fast16_t
atomic_uint_fast32_t
atomic_uint_fast64_t
atomic_uint_fast8_t
atomic_uint_least16_t
atomic_uint_least32_t
atomic_uint_least64_t
atomic_uint_least8_t
atomic_uintmax_t
atomic_uintptr_t
atomic_ullong
atomic_ulong
atomic_ushort
~~ATOMIC_VAR_INIT~~
atomic_wchar_t
ATOMIC_WCHAR_T_LOCK_FREE
~~_Bool~~
~~__bool_true_false_are_defined~~
cnd_broadcast
cnd_destroy
cnd_init
cnd_signal
cnd_t
cnd_timedwait
cnd_wait
~~_Complex~~ __concurrent__
~~_Complex_I~~ __CORE_
__CORE__
__CORE_ALIAS_OVERWRITES__
__CORE_CHAR16_IS_TYPE__
__CORE_CHAR32_IS_TYPE__
__CORE_CHAR8_IS_TYPE__
__CORE_NO_ATOMICS__
__CORE_NO_COMPLEX__
__CORE_NO_VLA__
__CORE_PTRDIFF_IS_TYPE__
__CORE_SIZE_IS_TYPE__
__CORE_VERSION__
__CORE_VERSION_COMPLEX_H__
__CORE_VERSION_FENV_H__
__CORE_VERSION_INTTYPES_H__
__CORE_VERSION_MATH_H__
__CORE_VERSION_STDATOMIC_H__
__CORE_VERSION_STDINT_H__
__CORE_VERSION_STDLIB_H__
__CORE_VERSION_STRING_H__
__CORE_VERSION_TIME_H__
__CORE_VERSION_UCHAR_H__
__CORE_VERSION_WCHAR_H__
__CORE_WCHAR_IS_TYPE__
__CORE_WEAK_THREEWAY_COMPARISON__
__cplusplus
__DATE__
__deprecated__
EDOM
EILSEQ
EOF
EOL
ERANGE
__evaluates__
_Exit
EXIT_FAILURE
EXIT_SUCCESS
__fallthrough__
FE_ALL_EXCEPT
FE_DFL_ENV
FE_DIVBYZERO
FE_DOWNWARD
FE_INEXACT
FE_INVALID
FE_OVERFLOW
_fetch
_fetch_explicit

FE_TONEAREST
FE_TONEARESTFROMZERO
FE_TOWARDZERO
FE_UNDERFLOW
FE_UPWARD
__FILE__
~~__func__~~ __free__
~~_Generic~~ __func__
~~_Imaginary~~ __idempotent__
~~_Imaginary_I~~ __independent__
INT16_C
INT16_MAX
INT16_MIN
int16_t
INT32_C
INT32_MAX
INT32_MIN
int32_t
INT64_C
INT64_MAX
INT64_MIN
int64_t
INT8_C
INT8_MAX
INT8_MIN
int8_t
INT_BITFIELD_MAX
int_fast16_t
int_fast32_t
int_fast64_t
int_fast8_t
int_least16_t
int_least32_t
int_least64_t
int_least8_t
INT_LEAST_WIDTH_MAX
INT_MAX
INTMAX_C
INTMAX_MAX
INTMAX_MIN
intmax_t
INT_MIN
INTPTR_MAX
INTPTR_MIN
intptr_t
_IOFBF
_IOLBF
_IONBF
isalnum
isalpha
isblank
iscntrl
iscomplex
isconstant
isdigit
isextended
isfinite
isfloating
isfunction
isgraph
isgreater
isgreaterequal
isice
isinf
isinteger
isless
islessequal
islessgreater
islower
isnan
isnarrow
isnormal
isnull
ispointer
isprint
ispunct
isreal
issame
issigned
isspace
isstrong
issymmetric
isunordered
isunsigned
isupper
isvla
iswalnum
iswalpha
iswblank
iswcntrl
iswctype
iswdigit
iswgraph
iswide
iswlower
iswprint
iswpunct
iswspace
iswupper
iswxdigit
isxdigit
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
__LINE__
__maybe_unused__
memccpy
memchr
memcmp
memcpy
~~memcpy_s~~
memmove
~~memmove_s~~
memory_order
memory_order_acq_rel
memory_order_acquire
memory_order_consume
memory_order_relaxed
memory_order_release
memory_order_seq_cst
memset
~~memset_s~~ __modifies__
mtx_destroy
mtx_init
mtx_lock
mtx_plain
mtx_recursive
mtx_t
mtx_timed
mtx_timedlock
mtx_trylock
mtx_unlock
__noalias__
__nodiscard__
__noleak__
_Noreturn
_Pragma
PRId32
PRId64

PRIdFAST32
PRIdFAST64
PRIdLEAST32
PRIdLEAST64
PRIdMAX
PRIdPTR
PRIi32
PRIi64
PRIiFAST32
PRIiFAST64
PRIiLEAST32
PRIiLEAST64
PRIiMAX
PRIiPTR
PRIo32
PRIo64
PRIoFAST32
PRIoFAST64
PRIoLEAST32
PRIoLEAST64
PRIoMAX
PRIoPTR
PRIu32
PRIu64
PRIuFAST32
PRIuFAST64
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIuPTR
PRIX32
PRIX64
PRIXFAST32
PRIXFAST64
PRIXLEAST32
PRIXLEAST64
PRIXMAX
PRIXPTR
__realloc__
__reentrant__
__reinterpret__
SCNdMAX
SCNdPTR
SCNiMAX
SCNiPTR
SCNoMAX
SCNoPTR
SCNuMAX
SCNuPTR
SCNxMAX
SCNxPTR
SIGABRT
SIG_ATOMIC_MAX
SIG_ATOMIC_MIN
SIG_ATOMIC_WIDTH
SIG_DFL
SIG_ERR
SIGFPE
SIG_IGN
SIGILL
SIGINT
SIGSEGV
SIGTERM
~~_Static_assert~~ __state_conserving__
__state_invariant__
__stateless__
__state_transparent__
__STDC__
__STDC_ANALYZABLE__
__STDC_HOSTED__
__STDC_IEC_559__
__STDC_IEC_559_COMPLEX__
__STDC_ISO_10646__
__STDC_LIB_EXT1__
__STDC_MB_MIGHT_NEQ_WC__
__STDC_NO_ATOMICS__
__STDC_NO_COMPLEX__
__STDC_NO_THREADS__
__STDC_NO_VLA__
__STDC_UTF_16__
__STDC_UTF_32__
__STDC_VERSION__
__STDC_VERSION_FENV_H__
~~__STDC_VERSION_MATH_H__~~
__STDC_VERSION_STDINT_H__
__STDC_VERSION_STDLIB_H__
__STDC_VERSION_TGMATH_H__
__STDC_VERSION_TIME_H__
__STDC_WANT_LIB_EXT1__
strcat
~~strcat_s~~
strchr
strcmp
strcoll
strcpy
~~strcpy_s~~
strcspn
strdup
strerror
~~strerrorlen_s~~
~~strerror_s~~
strftime
strlen
strncat
~~strncat_s~~
strncmp
strncpy
~~strncpy_s~~
strndup
~~strnlen_s~~ strong_equality
strong_ordering
strpbrk
strrchr
strspn
strstr
strtod
strtof
~~strtoimax~~
strtok
~~strtok_s~~
strtol
strtold
strtoll
strtoul
strtoull
~~strtoumax~~
struct
strxfrm
thrd_busy
thrd_create
thrd_current
thrd_detach
thrd_equal
thrd_error
thrd_exit
thrd_join
thrd_nomem
thrd_sleep
thrd_start_t
thrd_success
thrd_t
thrd_timedout

thrd_yield
~~_Thread_local~~
__TIME__
TIME_UTC
tocompare
tohighest
tolower
tolowest
tonullptr
toone
tostrong
toswitch
totext
totext_alt
totext_char
totext_exp
totext_fixed
totext_sign
totext_up
toupper
tovoidptr
towctrans
towlower
towupper
tozero
tss_create
tss_delete
tss_dtor_t
tss_get
tss_set
tss_t
UINT16_C
UINT16_MAX
uint16_t
UINT32_C
UINT32_MAX
uint32_t
UINT64_C
UINT64_MAX
uint64_t
UINT8_C
UINT8_MAX
uint8_t
uint_fast16_t
uint_fast32_t
uint_fast64_t
uint_fast8_t
uint_least16_t
uint_least32_t
uint_least64_t
uint_least8_t
UINT_MAX
UINTMAX_C
UINTMAX_MAX
uintmax_t
UINTPTR_MAX
uintptr_t
~~__VA_ARGS__~~
~~wcscat~~
~~wcscat_s~~
~~wcschr~~
~~wcscmp~~
~~wcscoll~~
~~wcscpy~~
~~wcscpy_s~~ __unsequenced__
~~wcscspn~~ __VA_ARGS__
wcsftime
~~wcslen~~
~~wcsncat~~
~~wcsncat_s~~
~~wcsncmp~~
~~wcsncpy~~
~~wcsncpy_s~~
~~wcsnlen_s~~
~~wcspbrk~~
~~wcsrchr~~
wcsrtombs
~~wcsrtombs_s~~
~~wcsspn~~
~~wcsstr~~
~~wcstod~~
~~wcstof~~
~~wcstoimax~~
~~wcstok~~
~~wcstok_s~~
~~wcstol~~
~~wcstold~~
~~wcstoll~~
wcstombs
~~wcstombs_s~~
~~wcstoul~~
~~wcstoull~~
~~wcstoumax~~
~~wcsxfrm~~ __writethrough__

J.6.2 Particular identifiers or keywords

1 The following ~~734~~ 538 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

For features that are obsolescent, see below.

abort
~~abort_handler_s~~
abs
acos
~~acosf~~
acosh
~~acoshf~~
~~acoshl~~ address_independent
~~acosl~~ alias
alignas
aligned_alloc
alignof
and
and_eq
asctime
asctime_r
~~asctime_s~~
asin
~~asinf~~
asinh
~~asinhf~~
~~asinhl~~
~~asinl~~
assert
atan
atan2
~~atan2f~~
~~atan2l~~
~~atanf~~
atanh
~~atanhf~~
~~atanhl~~
~~atanl~~
atexit
atof
atoi
atol
atoll
at_quick_exit
auto
bitand
bitor
bool
BOOL_MAX
BOOL_WIDTH
break
bsearch
~~bsearch_s~~
btowc
BUFSIZ
c16rtomb
c32rtomb
~~cabs~~
~~cabsf~~
~~cabsl~~
~~cacos~~
~~cacosf~~
~~cacosh~~
~~cacoshf~~
~~cacoshl~~

~~cacosl~~ c8rtomb
calloc
call_once
carg
~~cargf~~
~~cargl~~
case
~~casin~~
~~casinf~~
~~casinh~~
~~casinhf~~
~~casinhl~~
~~casinl~~
~~catan~~
~~catanf~~
~~catanh~~
~~catanhf~~
~~catanhl~~
~~catanl~~
cbrt
~~cbrtf~~
~~cbrtl~~
~~ccos~~
~~ccosf~~
~~ccosh~~
~~ccoshf~~
~~ccoshl~~
~~ccosl~~
ceil
~~ceilf~~
~~ceill~~
~~cerf~~
~~cerfc~~
~~cexp~~
~~cexp2~~
~~cexpf~~
~~cexpl~~
~~cexpm1~~
char
char16_t
char32_t
char8_t
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHAR_WIDTH
cimag
~~cimagf~~
~~cimagl~~
clearerr
~~clgamma~~
clock
CLOCKS_PER_SEC
clock_t
~~clog~~
~~clog10~~
~~clog1p~~
~~clog2~~
~~clogf~~
~~clogl~~
CMPLX
CMPLXF
CMPLXL
compare_t
compl
~~complex~~ complex_type
~~conj~~ complex_value
~~conjf~~ concurrent
~~conjl~~ conj
const
~~constraint_handler_t~~
constexpr
continue
copysign
~~copysignf~~
~~copysignl~~ CORE
cos
~~cosf~~
cosh
~~coshf~~
~~coshl~~
~~cosl~~
~~cpow~~
~~cpowf~~
~~cpowl~~
cproj
~~cprojf~~
~~cprojl~~
creal
~~crealf~~
~~creall~~
~~csin~~
~~csinf~~
~~csinh~~
~~csinhf~~
~~csinhl~~
~~csinl~~
~~csqrt~~
~~csqrtf~~
~~csqrtl~~
~~ctan~~
~~ctanf~~
~~ctanh~~
~~ctanhf~~
~~ctanhl~~
~~ctanl~~
~~ctgamma~~
ctime
ctime_r
~~ctime_s~~
currency_symbol
CX_LIMITED_RANGE
DBL_DECIMAL_DIG
DBL_DIG
DBL_EPSILON
DBL_HAS_SUBNORM
DBL_MANT_DIG
DBL_MAX
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN
DBL_MIN_10_EXP
DBL_MIN_EXP
DBL_NORM_MAX
DBL_TRUE_MIN
~~DECIMAL_DIG~~
decimal_point
decltype
DEFAULT
define
defined
deprecated
difftime
div
~~div_t~~
do
double
double_t
elif
else
endif
enum
~~erf~~
~~erfc~~ environ
~~erfcf~~ equal
~~erfcl~~ equivalent
~~erff~~ erf
~~erfl~~ erfc
errno
~~errno_t~~
error
evaluates
exit
exp
exp2
~~exp2f~~
~~exp2l~~
~~expf~~
~~expl~~
expm1
~~expm1f~~
~~expm1l~~
extern
fabs
~~fabsf~~
~~fabsl~~
fallthrough
false
fclose
fdim
~~fdimf~~
~~fdiml~~
feclearexcept
fegetenv
fegetexceptflag
fegetround
feholdexcept
fenv
FENV_ACCESS
fenv_t
feof
feraiseexcept
ferror
fesetenv
fesetexceptflag
fesetround
fetestexcept
feupdateenv
fexcept_t
fflush
fgetc
fgetpos
fgets
fgetwc
fgetws
FILE
FILENAME_MAX
float
floating_value
float_t
floor
floorf
floorl
FLT_DECIMAL_DIG
FLT_DIG
FLT_EPSILON
FLT_EVAL_METHOD
FLT_HAS_SUBNORM
FLT_MANT_DIG
FLT_MAX
FLT_MAX_10_EXP
FLT_MAX_EXP
FLT_MIN
FLT_MIN_10_EXP
FLT_MIN_EXP
FLT_NORM_MAX
FLT_RADIX
FLT_ROUNDS
FLT_TRUE_MIN
fma
~~fmaf~~
~~fmal~~
fmax
~~fmaxf~~
~~fmaxl~~
fmin
~~fminf~~
~~fminl~~
fmod
~~fmodf~~
~~fmodl~~
fopen
FOPEN_MAX
~~fopen_s~~
for
fpclassify
FP_CONTRACT
FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL
FP_ILOGB0
FP_ILOGBNAN
FP_INFINITE
FP_NAN
FP_NORMAL
fpos_t
fprintf
~~fprintf_s~~
FP_SUBNORMAL
fputc
fputs
fputwc
fputws
FP_ZERO
frac_digits
fread
free
freopen
~~freopen_s~~
frexp
~~frexpf~~
~~frexpl~~
fscanf
~~fscanf_s~~
fseek
fsetpos
ftell
FUNCTION_ATTRIBUTE
fwide
fwprintf
~~fwprintf_s~~
fwrite
fwscanf
~~fwscanf_s~~
generic_selection
generic_type
generic_value
getc
getchar
getenv
~~getenv_s~~
~~gets~~
~~gets_s~~
getwc

getwchar
gmtime
gmtime_r
~~gmtime_s~~
goto
greater
grouping
HUGE_VAL
HUGE_VALF
HUGE_VALL
hypot
~~hypotf~~
~~hypotl~~
~~I~~ idempotent
if
ifdef
ifndef
~~ignore_handler_s~~
ilogb
~~ilogbf~~
~~ilogbl~~
~~imaginary~~
~~imaxabs~~
~~imaxdiv~~
~~imaxdiv_t~~
imaginary_value
include
independent
INFINITY
inline
INT16_WIDTH
INT32_WIDTH
INT64_WIDTH
INT8_WIDTH
int_curr_symbol
int_frac_digits
INTMAX_WIDTH
int_n_cs_precedes
int_n_sep_by_space
int_n_sign_posn
int_p_cs_precedes
int_p_sep_by_space
int_p_sign_posn
INTPTR_WIDTH
INT_WIDTH
intwidth
jmp_buf
kill_dependency
~~labs~~
lconv
LDBL_DECIMAL_DIG
LDBL_DIG
LDBL_EPSILON
LDBL_HAS_SUBNORM
LDBL_MANT_DIG
LDBL_MAX
LDBL_MAX_10_EXP
LDBL_MAX_EXP
LDBL_MIN
LDBL_MIN_10_EXP
LDBL_MIN_EXP
LDBL_NORM_MAX
LDBL_TRUE_MIN
ldexp
~~ldexpf~~
~~ldexpl~~
~~ldiv~~
~~ldiv_t~~ less
lgamma
~~lgammaf~~
~~lgammal~~
line
~~llabs~~
~~lldiv~~
~~lldiv_t~~
LLONG_MAX
LLONG_MIN
LLONG_WIDTH
llrint
~~llrintf~~
~~llrintl~~
llround
~~llroundf~~
~~llroundl~~ locale
localeconv
localtime
localtime_r
~~localtime_s~~
log
log10
~~log10f~~
~~log10l~~
log1p
~~log1pf~~
~~log1pl~~
log2
~~log2f~~
~~log2l~~
logb
~~logbf~~
~~logbl~~
~~logf~~
~~logl~~
long
longjmp
LONG_MAX
LONG_MIN
LONG_WIDTH
lrint
~~lrintf~~
~~lrintl~~
lround
~~lroundf~~
~~lroundl~~
L_tmpnam
~~L_tmpnam_s~~
main
malloc
MATH_ERREXCEPT
math_errhandling
MATH_ERRNO
math_pdiff
max
max_align_t
maybe_unused
MB_CUR_MAX
mblen
MB_LEN_MAX
mbrlen
mbrtoc16
mbrtoc32
mbrtoc8
mbrtowc
mbsinit
mbsrtowcs
~~mbsrtowcs_s~~
mbstate_t
mbstowcs
~~mbstowcs_s~~
mbtowc
min
mktime
modf
~~modff~~
~~modfl~~ modifies
mon_decimal_point
mon_grouping
mon_thousands_sep
nan
nanf
nanl
n_cs_precedes
NDEBUG
nearbyint
~~nearbyintf~~
~~nearbyintl~~
negative_sign
nextafter
~~nextafterf~~
~~nextafterl~~
nexttoward
~~nexttowardf~~
~~nexttowardl~~ noalias
nodiscard
noleak
nonequivalent
noreturn
not
not_eq
n_sep_by_space
n_sign_posn
~~NULL~~ nullptr
nullptr_t
OFF
offsetof
ON
once_flag
~~ONCE_FLAG_INIT~~
or
or_eq
partial_ordering
p_cs_precedes
perror
positive_sign
pow
~~powf~~
~~powl~~
pragma
printf
~~printf_s~~
p_sep_by_space
p_sign_posn
PTRDIFF_MAX
PTRDIFF_MIN
ptrdiff_t
PTRDIFF_WIDTH
putc
putchar
puts
putwc
putwchar
qsort
~~qsort_s~~
quick_exit
raise
rand
RAND_MAX
realloc
~~register~~ real_type
~~remainder~~ real_value
~~remainderf~~ reentrant
~~remainderl~~ reinterpret
remainder
remove
remquo
~~remquof~~
~~remquol~~
rename
~~restrict~~
return
rewind
rint
~~rintf~~
~~rintl~~
round
~~roundf~~
~~roundl~~
RSIZE_MAX
~~rsize_t~~
~~scalbln~~
~~scalblnf~~
~~scalblnl~~
scalbn
~~scalbnf~~
~~scalbnl~~
scanf
~~scanf_s~~
SCHAR_MAX
SCHAR_MIN
SCHAR_WIDTH
search
SEEK_CUR
SEEK_END
SEEK_SET
setbuf
set_constraint_handler_s
setjmp
setlocale
setvbuf
short
SHRT_MAX
SHRT_MIN
SHRT_WIDTH
sig_atomic_t
signal
signbit
signed
sin
~~sinf~~
sinh
~~sinhf~~
~~sinhl~~
~~sinl~~
SIZE_MAX
sizeof
size_t
SIZE_WIDTH
snprintf
~~snprintf_s~~
~~snwprintf_s~~ sort
sprintf
~~sprintf_s~~
sqrt
~~sqrtf~~
~~sqrtl~~
srand
sscanf
~~sscanf_s~~
state_conserving
state_invariant
stateless
state_transparent
static

static_assert
std
STDC
stderr
stdin
stdout
switch
swprintf
~~swprintf_s~~
swscanf
~~swscanf_s~~
system
tan
~~tanf~~
tanh
~~tanhf~~
~~tanhl~~
~~tanl~~
tgamma
~~tgammaf~~
~~tgammal~~
thousands_sep
thread_local
time
timespec
timespec_get
time_t
tm
tm_hour
tm_isdst
tm_mday
tm_min
tm_mon
tmpfile
~~tmpfile_s~~
TMP_MAX
TMP_MAX_S
tmpnam
~~tmpnam_s~~
tm_sec
tm_wday
tm_yday
tm_year
true
trunc
~~truncf~~
~~truncl~~
TSS_DTOR_ITERATIONS
tv_nsec
tv_sec
typedef
UCHAR_MAX
UCHAR_WIDTH
UINT16_WIDTH
UINT32_WIDTH
UINT64_WIDTH
UINT8_WIDTH
UINTMAX_WIDTH
UINTPTR_WIDTH
UINT_WIDTH
uintwidth
ULLONG_MAX
ULLONG_WIDTH
ULONG_MAX
ULONG_WIDTH
undef
unequal
ungetc
ungetwc
union
unordered
unsequenced
unsigned
USHRT_MAX
USHRT_WIDTH
va_arg
va_copy
va_end
va_list
va_start
vfprintf
~~vfprintf_s~~
vfscanf
~~vfscanf_s~~
vfwprintf
~~vfwprintf_s~~
vfwscanf
~~vfwscanf_s~~
void
volatile
vprintf
~~vprintf_s~~
vscanf
~~vscanf_s~~
vsnprintf
~~vsnprintf_s~~
~~vsnwprintf_s~~
vsprintf
~~vsprintf_s~~
vsscanf
~~vsscanf_s~~
vswprintf
~~vswprintf_s~~
vswscanf
~~vswscanf_s~~
vwprintf
~~vwprintf_s~~
vwscanf
~~vwscanf_s~~
WCHAR_MAX
WCHAR_MIN
wchar_t
WCHAR_WIDTH
wcrtomb
~~wcrtomb_s~~
wctob
wctomb
~~wctomb_s~~
wctrans
wctrans_t
wctype
wctype_t
weak_equality
weak_ordering
WEOF
while
WINT_MAX
WINT_MIN
wint_t
WINT_WIDTH
wmemchr
wmemcmp
wmemcpy
~~wmemcpy_s~~
wmemmove
~~wmemmove_s~~
wmemset
wprintf
~~wprintf_s~~ writethrough
wscanf
~~wscanf_s~~
xor
xor_eq

J.6.3 Obsolete identifiers or keywords

1 The following 236 identifiers or keywords are obsolescent and should not be used by applications because they may be removed in future versions of this standard.

acosf
acoshf
acoshl
acosl
_Alignas
__alignas_is_defined
_Alignof
__alignof_is_defined
asinf
asinhf
asinhl
asinl
atan2f
atan2l
atanf
atanhf
atanhl
atanl
_Atomic
ATOMIC_FLAG_INIT
ATOMIC_VAR_INIT
_Bool
__bool_true_false_are_defined
cabs
cabsf
cabsl
cacos
cacosf
cacosh
cacoshf
cacoshl
cacosl
cargf
cargl
casin
casinf
casinh
casinhf
casinhl
casinl
catan
catanf
catanh
catanhf
catanhl
catanl
cbrtf
cbrtl
ccos
ccosf
ccosh
ccoshf
ccoshl
ccosl
ceilf
ceill
cexp
cexpf
cexpl
cimagf
cimagl
clog
clog10
clog1p
clogf
clogl
_Complex
complex
_Complex_I
conjf
conjl
copysignf
copysignl
cosf
coshf
coshl
cosl
cpow
cpowf
cpowl
cprojf
cprojl
crealf
creall

```
csin
csinf
csinh
csinhf
csinhl
csinl
csqrt
csqrtf
csqrtl
ctan
ctanf
ctanh
ctanhf
ctanhl
ctanl
DECIMAL_DIG
div_t
erfcf
erfcl
erff
erfl
exp2f
exp2l
expf
expl
expm1f
expm1l
fabsf
fabsl
fdimf
fdiml
fmaf
fmal
fmaxf
fmaxl
fminf
fminl
fmodf
fmodl
frexpf
frexpl
_Generic
gets
hypotf
hypotl
I
ilogbf
ilogbl
_Imaginary
imaginary
_Imaginary_I
imaxabs
imaxdiv
imaxdiv_t
labs
ldexpf
ldexpl
ldiv
ldiv_t
lgammaf
lgammal
llabs
lldiv
lldiv_t
llrintf
llrintl
llroundf
llroundl
log10f
log10l
log1pf
log1pl
log2f
log2l
logbf
logbl
logf
logl
lrintf
lrintl
lroundf
lroundl
modff
modfl
nearbyintf
nearbyintl
nextafterf
nextafterl
nexttowardf
nexttowardl
NULL
ONCE_FLAG_INIT
powf
powl
register
remainderf
remainderl
remquof
remquol
restrict
rintf
rintl
roundf
roundl
scalbln
scalblnf
scalblnl
scalbnf
scalbnl
sinf
sinhf
sinhl
sinl
sqrtf
sqrtl
_Static_assert
strtoimax
strtoumax
tanf
tanhf
tanhl
tanl
tgammaf
tgammal
_Thread_local
truncf
truncl
wcscat
wcschr
wcscmp
wcscoll
wcscpy
wcscspn
wcslen
wcsncat
wcsncmp
wcsncpy
wcspbrk
wcsrchr
wcsspn
wcsstr
wcstod
wcstof
wcstoimax
wcstok
wcstol
wcstold
wcstoll
wcstoul
wcstoull
wcstoumax
wcsxfrm
```

Annex K
(removed)
Bounds-checking interfaces

For C, this annex describes a large set of extensions that are only scarcely implemented on real platforms and have a lot of issues. It has not found consensus in the C community and has never been adapted to C++. Therefore these interfaces are not part of the C/C++ core.

Annex L
(normative)
Analyzability

L.1 Scope

1 This annex specifies optional behavior that can aid in the analyzability of C programs.

2 An implementation that defines `__STDC_ANALYZABLE__` shall conform to the specifications in this annex.[483)]

L.2 Definitions

L.2.1

1 **out-of-bounds store**

an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared **`volatile`**, fetch) one or more bytes that lie outside the bounds permitted by this document.

L.2.2

1 **bounded undefined behavior**

undefined behavior (3.4.3) that does not perform an out-of-bounds store.

2 **Note 1 to entry:** The behavior might perform a trap.

3 **Note 2 to entry:** Any values produced or stored might be indeterminate values.

L.2.3

1 **critical undefined behavior**

undefined behavior that is not bounded undefined behavior.

2 **Note 1 to entry:** The behavior might perform an out-of-bounds store or perform a trap.

L.3 Requirements

1 If the program performs a trap (3.22.5), the implementation is permitted to invoke a runtime-constraint handler. Any such semantics are implementation-defined.

2 All undefined behavior shall be limited to bounded undefined behavior, except for the following which are permitted to result in critical undefined behavior:

— An object is referred to outside of its lifetime (6.2.4).

— A store is performed to an object that has two incompatible declarations (6.2.7),

— A pointer is used to call a function whose type is not compatible with the referenced type (6.2.7, 6.3.2.3, 6.5.2.2).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— The program attempts to modify a string literal (6.4.5).

— The operand of the unary $*$ operator has an invalid value (6.5.3.2).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary $*$ operator that is evaluated (6.5.6).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

483) Implementations that do not define `__STDC_ANALYZABLE__` are not required to conform to these specifications.

— An argument to a function or macro defined in the standard library has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

— The **longjmp** function is called with a **jmp_buf** argument where the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument is nonexistent, or the invocation was from another thread of execution, or the function containing the invocation has terminated execution in the interim, or the invocation was within the scope of an identifier with variably modified type and execution has left that scope in the interim (7.13.2.1).

— The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.22.4).

— A string or wide string utility function accesses an array beyond the end of an object (7.24.1.1~~, ??~~).

Annex M
(informative)
Change History

M.1 Fifth Edition

1 Major changes in this core specification (`__CORE_VERSION__` *202005*L) include the following changes that are scheduled to appear in C2x:

— remove obsolete sign representations and integer width constraints

— remove function definitions that don't provide prototype

— allow nameless parameter declarations

— added a one-argument version of ~~`_Static_assert`~~ `static_assert`, make it a keyword and deprecate the underscore-capital form

— support for function definitions with identifier lists has been removed

— harmonization with ISO/IEC 9945 (POSIX):

- extended month name formats for `strftime`
- integration of functions: `asctime_r`, `ctime_r`, `gmtime_r`, `localtime_r`, `memccpy`, `strdup`, `strndup`

— the macro `DECIMAL_DIG` is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added `deprecated`, `fallthrough`, `maybe_unused`, and `nodiscard` attributes

— added the `u8` character prefix

— change `bool`, `alignas`, `alignof` and `thread_local` to be keywords and deprecate the underscore-capital forms

— change `false` and `true` to keywords and make them type `bool`

— added a universal null pointer constant `nullptr`

Changes that are not yet included in the C standard are:

Change history

v1: WG14 document N2494

v2: WG14 document N2522, this version, diffmarks from "cmin"

- clarifications: lambdas and `longjmp`, `inline` and scope, type `char8_t`, generic selection, `core::unsequenced` attribute
- three-way comparison (spaceship operator)
- "initializer" construct for captures of lambdas
- a tool for textual representation of all basic types and arrays, `totext`
- more attributes for allocated storage, `core::free`, `core::realloc`, `core::noleak`
- an attribute for tracking (or not) of initializations `core::writethrough`
- `core::concurrent` attribute
- `constexpr` based on `core::concurrent` and annotation of some library calls with that specifier
- `for`-loop and `if` with variable definitions
- harmonization of tag names between C and C++
- hammonization of the requirements for Unicode as an internal encoding

M.2 Fourth Edition

1 There were no major changes in the fourth edition (`__STDC_VERSION__` `201710L`), only technical corrections and clarifications.

M.3 Third Edition

1 Major changes in the third edition (`__STDC_VERSION__` `201112L`) included:

— conditional (optional) features (including some that were previously mandatory)

— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (`<stdatomic.h>` and `<threads.h>`)

— additional floating-point characteristic macros (`<float.h>`)

— querying and specifying alignment of objects (`<stdalign.h>`, `<stdlib.h>`)

— Unicode characters and strings (`<uchar.h>`) (originally specified in ISO/IEC TR 19769:2004)

— type-generic expressions

— static assertions

— anonymous structures and unions

— no-return functions

— macros to create complex numbers (`<complex.h>`)

— support for opening files for exclusive access

— removed the **`gets`** function (`<stdio.h>`)

— added the **`aligned_alloc`**, **`at_quick_exit`**, and **`quick_exit`** functions (`<stdlib.h>`)

— (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731–1:2007)

— (conditional) support for analyzability

M.4 Second Edition

1 Major changes in the second edition (`__STDC_VERSION__` `199901L`) included:

— restricted character set support via digraphs and `<iso646.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)

— wide character library support in `<wchar.h>` and `<wctype.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)

— more precise aliasing rules via effective type

— restricted pointers

— variable length arrays

— flexible array members

— **`static`** and type qualifiers in parameter array declarators

— complex (and imaginary) support in `<complex.h>`

— type-generic math macros in `<tgmath.h>`

— the **`long long int`** type and library functions

— extended integer types

— increased minimum translation limits

— additional floating-point characteristics in <float.h>

— remove implicit **int**

— reliable integer division

— universal character names (\u and \U)

— extended identifiers

— hexadecimal floating-point constants and %a and %A **printf**/**scanf** conversion specifiers

— compound literals

— designated initializers

— // comments

— specified width integer types and corresponding library functions in <inttypes.h> and <stdint.h>

— remove implicit function declaration

— preprocessor arithmetic done in **intmax_t**/**uintmax_t**

— mixed declarations and statements

— new block scopes for selection and iteration statements

— integer constant type rules

— integer promotion rules

— macros with a variable number of arguments (**__VA_ARGS__**)

— the **vscanf** family of functions in <stdio.h> and <wchar.h>

— additional math library functions in <math.h>

— treatment of error conditions by math library functions (**math_errhandling**)

— floating-point environment access in <fenv.h>

— IEC 60559 (also known as IEC 559 or IEEE arithmetic) support

— trailing comma allowed in **enum** declaration

— %lf conversion specifier allowed in **printf**

— inline functions

— the **snprintf** family of functions in <stdio.h>

— boolean type in <stdbool.h>

— idempotent type qualifiers

— empty macro arguments

— new structure type compatibility rules (tag compatibility)

— additional predefined macro names

— **_Pragma** preprocessing operator

— standard pragmas

— `__func__` predefined identifier

— `va_copy` macro

— additional `strftime` conversion specifiers

— LIA compatibility annex

— deprecate `ungetc` at the beginning of a binary file

— remove deprecation of aliased array parameters

— conversion of array to pointer not limited to lvalues

— relaxed constraints on aggregate and union initialization

— relaxed restrictions on portable header names

— `return` without expression not permitted in function that returns a value (and vice versa)

M.5 First Edition, Amendment 1

1 Major changes in the amendment to the first edition (`__STDC_VERSION__ 199409L`) included:

— addition of the predefined `__STDC_VERSION__` macro

— restricted character set support via digraphs and `<iso646.h>`

— wide character library support in `<wchar.h>` and `<wctype.h>`

(no diff marks, here)

Bibliography

[1] ISO/IEC 646:1991, *Information technology* — ISO *7-bit coded character set for information interchange*.

[2] ISO/IEC 9945–2:1993, *Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities*.

[3] ISO/IEC TR 10176:2003, *Information technology — Guidelines for the preparation of programming language standards*.

[4] ISO/IEC 10967–1:2012, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*.

[5] ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

[6] ISO/IEC TR 24731–1:2007, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*.

[7] ANSI/IEEE 754–1985, *American National Standard for Binary Floating-Point Arithmetic*.

[8] ANSI/IEEE 854–1988, *American National Standard for Radix-Independent Floating-Point Arithmetic*.

[9] ANSI X3/TR–1–82 (1982), *American National Dictionary for Information Processing Systems*, Information Processing Systems Technical Report.

[10] "The C Reference Manual" by Dennis M. Ritchie, a version of which was published in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978). Copyright owned by AT&T.

[11] *1984 /usr/group Standard* by the /usr/group Standards Committee, Santa Clara, California, USA, November 1984.

Index