Programming languages –
A Provenance-aware memory object model for C

Technical Specification

Jens Gustedt[1], Peter Sewell[2], Kayvan Memarian[2], Victor B. F. Gomes[2], Martin Uecker[3]

[1]INRIA and ICube, Université de Strasbourg, France
[2]University of Cambridge, UK
[3]University Medical Center, Göttingen, Germany

ISO TC1/SC22/WG14
document number: **N3057**
document date: 2022-09-20

Changes:

2020-10-4: WG14 N2577, orginal version

2021-3-24: WG14 N2676

- Change in terminology. Instead of using the confusing "indeterminate value", change to "invalid value" for pointer values and to "indeterminate state" to describe certain object representations.

  Synchronized with WG14 N2668.

2022-6-15: WG14 N3005

- Integrate changes of documents that are agreed by WG14 for the new revision of ISO/IEC 9899:2023.

  N2861 Indeterminate Values and Trap Representations

  N2888 Exact-width Integer Type Interfaces

- As agreed by WG14, integrate N2889, "Pointers and integer types", and adapt the proposed text to refer to `uintptr_t` for abstract addresses.
- Numerous editorial improvements.

2023-01-29: WG14 N3057, this version

- make document ready for TDS ballot
- make page numbering consistent
- implement ISO style requirements

first two pages to be replaced by ISO

Contents

Foreword iv

Introduction v

1 Scope 1

2 Normative references 2

3 Terms and definitions 3

4 Specifications 4
4.1 General . . . 4
4.2 Provenance-aware object models . . . 4
4.3 Operations on pointers . . . 8

Annex A (informative) Examples 10
A.1 General . . . 10
A.2 Basic pointer provenance . . . 10
A.3 Tracking provenance through integers . . . 11
A.4 Operations on pointer values and representations . . . 12
A.5 Implications of provenance semantics for optimizations . . . 19
A.6 Testing the example behavior in Cerberus . . . 24
A.7 Testing the example behavior in mainstream C implementations . . . 24

Annex B (informative) Detailed semantics 26
B.1 General . . . 26
B.2 The PNVI-ae-udi, PNVI-ae, PNVI-plain, and PVI semantics . . . 26
B.3 The memory object model state . . . 27

Annex C (normative) Differences to ISO/IEC 9899:2018 35

Bibliography 126

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see https://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

This document is divided into several subdivisions:

- preliminary elements (Clauses 1–3);
- The specification of a provenance-based memory object model of the C programming language (Clause 4);
- A set of executable examples that illustrate properties of this model (Annex A);
- A description of detailed semantics of this model in closed mathematical form (Annex B);
- A list of textual differences to ISO/IEC 9899:2018 that describe the memory object model that is specified by this document (Annex C).

The Working Group responsible for this document (WG 14) maintains a site on the World Wide Web at `http://www.open-std.org/JTC1/SC22/WG14/` containing ancillary information that may be of interest to some readers such as transitional documents and minutes of discussions that are the base for many of the decisions made during its preparation.

This document originates from various sources which are listed in the bibliography.

- The numbered clauses and Annex C have been prepared in the memory object model study group of WG 14. The main contributors are Victor B. F. Gomes, Jens Gustedt, Kayvan Memarian, Peter Sewell, and Martin Uecker.
- Annex A is derived from “C provenance semantics: examples.”
- Annex B is derived from “Exploring C semantics and pointer provenance.”

Programming languages –
A Provenance-aware memory object model for C.

1 Scope

This document defines a provenance-aware memory object model for the C programming language. It complements ISO/IEC 9899:2018, which defines most aspects of C but which leaves some aspects of its memory object model open. This document is therefore expressed as an extension to ISO/IEC 9899:2018, described in Section 4 and Annex C. Implementations that conform to this document shall behave as if these indicated differences to ISO/IEC 9899:2018 had been integrated into ISO/IEC 9899.

2 Normative references

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.

ISO/IEC 2382, *Information technology – Vocabulary*

ISO/IEC 9899:2018, *Programming languages – C*

ISO 80000–2, *Quantities and units — Part 2: Mathematics*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO/IEC 9899:2018, ISO 80000–2, and the following apply.

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/`

3.1

pointer provenance

provenance

an entity that is associated to a pointer value in the abstract machine, which is either empty, or the identity of a storage instance

3.2

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

4 Specifications

This document defines a provenance-aware memory object model for the C programming language. It complements ISO/IEC 9899:2018, which defines most aspects of C but which leaves some aspects of its memory object model open. This document is therefore expressed as an extension to ISO/IEC 9899:2018, described in Section 4 and Annex C. Implementations that conform to this document shall behave as if these indicated differences to ISO/IEC 9899:2018 had been integrated into ISO/IEC 9899.

4.1 General

In addition to the newly introduced terms (Clause 3), for the scope of this document the use of some terms of ISO/IEC 9899:2018 is replaced as listed in Annex C. Their meaning is as given there.

NOTE In particular the terms indeterminate value and trap representation are not used by this document. Instead, refined terms such as indeterminate representation and non-value representation are used as indicated.

Prior to that annex, Annex A and B provide executable examples and detailed semantics, respectively, for the different variants of the memory model that are discussed in this clause.

4.2 Provenance-aware object models

C pointer values are typically represented at runtime as simple concrete numeric values, but current implementations routinely exploit information about the provenance of pointers to reason that they cannot alias, and hence to justify optimizations. This document specifies a provenance semantics for simple cases of the construction and use of pointers.

ISO/IEC 9899:2018 does not explicitly speak to this: it is ambiguous there whether some programming idioms are allowed or not, and exactly what compiler alias analysis and optimization are allowed to do.

EXAMPLE For example, consider the well-known test case[1, 2, 5, 6, 8] below. This and many of the examples below are edge-cases, exploring the boundaries of what different semantic choices allow, and sometimes what behavior existing compilers exhibit; they are not all intended as desirable code idioms.

```
1  #include <stdio.h>
2  #include <string.h>
3  int y=2, x=1;
4  int main() {
5    int *p = &x + 1;
6    int *q = &y;
7    printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
8    if (memcmp(&p, &q, sizeof(p)) == 0) {
9      *p = 11;  // does this have undefined behaviour?
10     printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
11   }
12 }
```

Depending on the implementation, x and y can in some executions happen to be allocated in adjacent memory, in which case &x+1 and &y will have bitwise-identical representation values, the **memcmp** will succeed, and p (derived from a pointer to x) will have the same representation value as a pointer to a different object, y, at the point of the update *p=11.

This occurs in practice, e.g. with GCC 12.2.0 -O2 on some platforms. Its output of

```
x=1 y=2 *p=11 *q=2
```

suggests that the implementation is reasoning that *p does not alias with y or *q, and hence that the initial value of y=2 can be propagated to the final **printf**. The ICC implementation, e.g. ICC 19 -O2, also optimizes here (for a variant with x and y swapped), producing

```
x=1 y=2 *p=11 *q=11.
```

In contrast, Clang 15.0.2 -O2 just outputs the

```
x=1 y=11 *p=11 *q=11
```


that would be given by a concrete semantics.

Note that this example does not involve type-based alias analysis, and the outcome is not affected by implementation specific flags such as GCC or ICC's `-fno-strict-aliasing`. Moreover, the mere formation of the `&x+1` one-past pointer is explicitly permitted by ISO/IEC 9899:2018, and, because the `*p=11` access is guarded by the **`memcmp`** conditional check on the representation bytes of the pointer, it will not be attempted (and hence flag undefined behavior) in executions in which the two storage instances are not adjacent.

These GCC and ICC outcomes are not correct with respect to a concrete semantics, and so to make the existing behavior sound it is necessary for this program to be deemed to have undefined behavior.

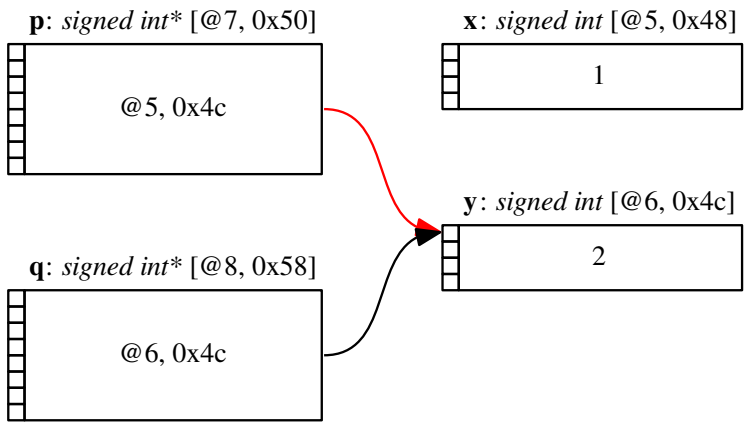
4.2.1 Basic provenance semantics for pointer values

For simple cases of the construction and use of pointers, capturing the basic intuition in a precise semantics is straightforward: a provenance is associated with every pointer value, identifying the original storage instance that the pointer is derived from:

- An abstract-machine pointer value shall be a pair (π, a), adding a provenance π, either @i where i is a storage instance ID, or the empty provenance @`empty`, to their concrete address a.
- On every creation of a storage instance (of objects with static, thread, automatic, and allocated storage duration), the abstract machine shall nondeterministically choose a fresh storage instance ID i (unique across the entire execution), and the resulting pointer value shall carry that single storage instance ID as its provenance @i.
- Provenance shall be preserved by pointer arithmetic that adds or subtracts an integer to a pointer.
- At any access via a pointer value, its byte address shall be consistent with its provenance; it has undefined behavior otherwise. In particular:
  - The access via a pointer value which has provenance a single storage instance ID @i shall be within the byte array of the corresponding original storage instance. That storage instance shall still be live.
  - All other accesses, including those via a pointer value with empty provenance, have undefined behavior.

NOTE This undefined behavior is what justifies optimization based on provenance alias analysis.

EXAMPLE Below is a provenance-semantics memory-state snapshot (from the Cerberus GUI) for the program example above, just before the invalid access via `p`, showing how the provenance mismatch makes it undefined behavior: at the attempted access via `p`, its pointer-value address 0x4c is not within the storage instance with the ID `@5` (the provenance of the allocation of `x`) of the provenance of the value of `p`.



NOTE This specification concerns the abstract machine as defined in ISO/IEC 9899:2018: implementations may rely on provenance in their alias analysis and optimization, but they are not expected to record or manipulate provenance at runtime; provenances

therefore do not necessarily have program-accessible runtime representations in the abstract machine. Nevertheless, dynamic or static analysis tools and non-standard or bug-finding-tool implementations may provide such access to provenance.

ISO/IEC 9899:2018 provides many other ways to construct and manipulate pointer values: casts to and from integers, copying with `memcpy`, manipulation of their representation bytes, type punning, I/O, copying with `realloc`, and constructing pointer values that embody knowledge established from linking.

This document specifies the

provenance not via integers
exposed-address
user-disambiguation

model, abbreviated as **PNVI-ae-udi**. This model is a refinement of two other, less restrictive models:

- **PNVI-plain** is a semantics that tracks provenance via pointer values but not via integers. At integer-to-pointer cast points, the given address shall point within a live storage instance and, if so, the corresponding provenance is recreated.

- **PNVI-ae (PNVI exposed-address)** is a variant of PNVI that allows integer-to-pointer casts to recreate provenance only for storage instances that have previously been exposed. Here, a storage instance is said to be exposed by a conversion to an integer type of a pointer to it, or by a read (at non-pointer type) of the representation of the pointer, or by a formatted output of the pointer using the format specifier `%p`.

- **PNVI-ae-udi (PNVI exposed-address user-disambiguation)** is a further refinement of PNVI-ae to support roundtrip casts, from pointer to integer and back, of pointers that are one-past a storage instance. In some circumstances such pointers can be ambiguous, referring either to one-past one storage instance or to the start of another adjacent storage instance. PNVI-ae-udi permits such ambiguity until some use of the pointer resolves it.

4.2.2 Storage instance IDs

An addressable storage instance is the byte array that is created when either an object starts its lifetime (for static, automatic and thread storage duration) or an allocation function is called (`malloc`, `calloc` etc). In addition to the abstract address, addressable storage instances shall have a unique ID throughout the whole execution.

NOTE There are also storage instances that are not addressable, namely for `register` variables, but since provenance needs pointers, these play no role in the following and are not discussed further.

Once the lifetime of an addressable storage instance ends, the implementation may attribute the same abstract address to a storage instance that is created thereafter, but never the same ID.

4.2.3 Provenance of pointers

The provenance of a valid pointer shall be the ID of the specific storage instance to which the pointer refers (or one-past).

NOTE As defined above, this ID is a genuine component of the pointer value in C's abstract machine, but it does not necessarily take part in the object representation of the pointer; in general the provenance of a pointer is not directly observable.

Valid pointers shall keep provenance to the encapsulating storage instance of the referred object. When the storage instance dies (falls out of scope, the thread ends, or `free` is called) the pointer value becomes invalid, and any stored representations of the pointer value become indeterminate.

Some valid pointers that are obtained by other means than by the unary & operator or as the return value of storage allocation functions can temporarily have two provenances; these are discussed in 4.2.6. Otherwise, valid pointers shall have a unique provenance.

4.2.4 Abstract address

The concept of abstract address lifts the implementation-defined mapping required for pointer-to-integer conversions, to the level of the memory model.

- Each byte of a storage instance shall have an abstract address, which is a positive integer of type `uintptr_t` that is constant during the whole lifetime of the storage instance.
- Abstract addresses shall be increasing within a storage instance.
- Storage instances shall be strictly ordered by the induced order of their abstract addresses.
- Storage instances and the ranges of their abstract addresses shall not overlap.

An implementation that conforms to this document shall define the types `uintptr_t` and `intptr_t` (specified as optional in ISO/IEC 9899:2018). If necessary these types may be wider than `uintmax_t` and `intmax_t`.

The set of all abstract addresses is said to be the address space of the execution. No other ordering constraints than the above between any pair of storage instances are implied.

NOTE This means in particular that no syntactic features (declaration order) or runtime features (order of allocation) imply a specific relative position of storage instances in the address space.

This concept is completely independent of the object representation of pointers: it is up to any implementation to define the relation between the two in any way that suits best. In particular, the address offset between consecutive bytes does not need to be 1 (or any other constant). There can be bumps (corresponding to segments, for example) and strides, and address sharing on the boundary between the one-past pointer of one storage instance and the start address of the next.

Compared to ISO/IEC 9899:2018, the type `uintptr_t` is made mandatory, here. On "usual" architectures where `uintptr_t` exists already, the abstract address of a pointer value `p` is just `(uintptr_t)p`. Architectures that do not yet have `uintptr_t` may define an abstract address that is consistent with the other operations that they allow on pointers; this document allows this type to be wider than `uintmax_t`.

4.2.5 Pointer exposure and synthesis

Tracking provenance for the sake of aliasing analysis can fail if pointers can acquire an abstract address with an arbitrary provenance of which the implementation is not aware. With the above specifications for abstract addresses such a possibility stems from a leak of information about a storage instance `A`, if:

- the abstract address of `A` has been made known, or
- the object representation of a pointer to `A` is inspected.

In such a case it is said that `A` has been exposed. Exposure of a storage instance constitutes a side effect in the abstract state machine.

NOTE The latter guarantees sequencing and synchronization properties of allocations and deallocations even though the side effect itself might not be directly observable.

In ISO/IEC 9899:2018 there are only very restricted contexts that construct pointer values `p` that are not directly deduced from the address of a storage instance, i.e. either by using the unary `&` operator on a defined object or by using a pointer that is returned by an allocation function. In such a case, `p` is said to be synthesized; a storage instance of such a synthesized pointer shall have been previously exposed.

NOTE This ensures that all storage instances that have not been exposed can be subject to a rigorous alias analysis, whereas pointers to potentially exposed storage instance acquire a clear "warning label" that tell the implementation to be cautious about them.

Operations that expose storage instances and synthesize pointers are described in 4.3.

4.2.6 Ambiguous Provenance of synthesize pointers

There is one special case where a synthesized pointer `p` can have two different provenances, `A` and `B`, referred to as ambiguous provenance. This happens when:

- `p` is the end address (one past) pointer of a storage instance `A` and the start address of another storage instance `B`, and

- both storage instances A and B are exposed.

In such a situation, both A and B are valid choices for the provenance in future operations on the pointer. Only one or the other (but not both) provenance shall be used for a given synthesized pointer value.

4.3 Operations on pointers

In the following, additional requirements for pointer operations that require or produce provenance are described. If a pointer value has ambiguous provenance before a pointer operation that makes use of provenance, one of these provenances is determined and shall be used henceforth in all pointer operations that operate on the same valid pointer. Details and consequences of such a disambiguation relative to clauses 6.2.5 p20 and 6.3.2.3 p9 of ISO/IEC 9899:2018 are presented in Annex C.

4.3.1 Exposing and non-exposing operations

A storage instance is exposed once information from any valid pointer with this provenance has leaked into other parts of the program state. The following five operations in ISO/IEC 9899:2018 provide information about the address of a storage instance A.

- A pointer value to A is converted to integer.
- A pointer value is passed as second argument to a call to `tss_set`.
- `printf` (or similar) with %p is used to print a pointer value.
- A byte of a pointer representation is accessed directly.
- A byte of a pointer representation is written with `fwrite`.

They are said to expose such a storage instance. No other library function shall expose storage instances, unless a callback function specified by the application that does so is called implicitly. **NOTE** Function pointer arguments to functions `qsort` or `atexit` constitute such application callbacks.

This requirement implies that library functions that receive pointers do not leak information about these pointers into global state.

4.3.2 Synthesizing operations

Valid pointer values are said to be synthesized if they originate from one of the following operations.

- An integer value has been converted to a pointer value.
- The representation of a pointer has been stored with a formatted input function (`scanf` or similar functions with %p).
- The representation of a pointer has been assembled by partial stores to representation bytes.
- The representation of a pointer has been read, even partially, with `fread`.

For the first, the pointer value of such a synthesizing operation shall refer to a byte (or one-past) of a live storage instance that previously had been exposed. The provenance of the result is then that storage instance.

For the other three, the object representation shall be a representation of a valid address (or one past) of a live storage instance that has been previously exposed. An lvalue conversion of such a representation shall return the corresponding pointer value and shall attribute that storage instance as the provenance to the pointer value.

4.3.3 Copying operations

Storing a pointer value by assignment in an object representation does not expose the corresponding storage instance. If the pointer value is not synthesized the resulting pointer representation is not synthesized.

Library functions that copy all bytes of an object representation of a pointer shall behave consistently with respect to the abstract state machine's notion of provenance. That is, if they copy the object representation of a pointer, they shall transfer provenance information consistently to the target. They do not expose the corresponding storage instance and the written object representation is not synthesized if the source representation is not synthesized.

NOTE Library functions that successfully transfer provenance without exposure are for example `memcpy`, `realloc` or `atomic_compare_exchange_weak`.

Byte-wise copy in application code is special, here, because there is no tool to indicate a transfer of a pointer value including provenance to the implementation. Therefore such a byte copy uses exposure, that is a pointer value that is copied byte-wise is first exposed (because the bytes of the object representation are accessed) and then synthesized as before by lvalue conversion.

4.3.4 Pointer inquiry

Pointer inquiry by equality or relational operations does not expose storage instances. Two valid pointers shall be equal if their abstract addresses are the same, two null pointers shall be equal, and a valid pointer value and a null pointer shall be unequal. Relational operators between two valid pointer values shall be such that the two pointers have the same provenance. The result is the relative position of the abstract addresses.

In all other cases the behavior is undefined.

NOTE Note that provenance is not used for the equality relation on pointers; indeed two pointers that have different provenance where one is the end address of one storage instance and the other is the start address of another can compare equal. In contrast to that, using a relational operator with pointer operands of different provenance or where one pointer is null has undefined behavior.

4.3.5 Pointer arithmetic

Pointer arithmetic does not expose storage instances and a pointer result of such arithmetic is not synthesized unless the source operand is synthesized.

Pointer arithmetic by addition or subtraction of integers shall preserve provenance and the resulting pointer shall be within (or one-past) the corresponding storage instance.

Pointer difference shall only be applied to pointers with the same provenance and that refer to elements within the same array.

NOTE The former is necessary because the one-past element of an array can be the first element of another storage instance that just happens to follow in the address space. The latter is necessary because pointer difference is not accounted in bytes but in number of elements of an array.

Annex A
(informative)

Examples

A.1 General

This annex gives a sequence of small test programs, designed to illustrate the semantic design questions and consequences of this Technical Specification as concisely as possible. Some are "natural" examples, of desirable C code that are found in applications, but many are intentionally pathological or are corner cases, to explore just where the defined/undefined-behavior boundary is.

ISO/IEC 9899:2018 is a prose document, and thus is not executable as a test oracle. To explore these examples (and variations) interactively, one can use the Cerberus tool[9]. This automatically computes the sets of allowed behaviors for each, including the detection of undefined behavior in some cases.

Making the tests concise, to illustrate semantic questions, also means that most are not written to trigger interesting behavior of translators. Moreover, conventional implementations cannot and do not report all instances of undefined behavior. Hence, only in some cases is there anything to be learned from the experimental compiler behavior. Clause A.7 gives an overview of the outcome of test runs on commonly available implementations.

For any executable semantics or analysis tool, on the other hand, all the tests should have instructive outcomes.

In addition to the three variants of the provenance-not-via-integer (PNVI) model that were introduced in clause 4.2.1 – the main PNVI-ae-udi variant specified by this TS, and two others, a provenance-via-integers (PVI) model is discussed. This is done to illustrate, by contrast, the exact design of, and motivation for, PNVI-ae-udi.

A.2 Basic pointer provenance

A.2.1 Pointer values that are out-of-bounds by more than one

Consider the example below, where q is transiently (more than one-past) out of bounds but brought back into bounds before being used for access. With ISO/IEC 9899:2018 6.5.6p8, constructing such a pointer value is clearly stated to be undefined behavior. This can be captured using the provenance of the pointer value to determine the relevant bounds. There are cases where such pointer arithmetic would go wrong on some implementations (some now exotic), e.g. where pointer arithmetic subtraction overflows, or if the transient value is not aligned and only aligned values are representable at the particular pointer type, or for hardware that does bounds checking, or where pointer arithmetic might wrap at values less than the obvious word size (e.g. "near" or "huge" pointers on the 8086 processor architecture).

// cheri_03_ii.c

```
1  int x[2];
2  int *p = &x[0];
3  int *q = p + 11; // defined behaviour?
4  q = q - 10;
5  *q = 1;
```

A.2.2 Inter-object pointer arithmetic

The example in 4.2 relied on guessing (and then checking) the offset between two storage instances. What if one instead calculates the offset, with pointer subtraction – enables that to move between objects, as below?

// pointer_offset_from_ptr_subtraction_global_xy.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stddef.h>
4  int x=1, y=2;
5  int main() {
6    int *p = &x;
```



```
  int *q = &y;
  ptrdiff_t offset = q - p;
  int *r = p + offset;
  if (memcmp(&r, &q, sizeof(r)) == 0) {
    *r = 11; // is this free of UB?
    printf("y=%d *q=%d *r=%d\n",y,*q,*r);
  }
}
```

In ISO/IEC 9899:2018, the pointer arithmetic q-p has undefined behavior as it is a pointer subtraction between pointers to different objects. So the above program clearly has undefined behavior.

Even if the calculation were replaced with an educated guess about the offset value, in the model specified here either the expression p+offset (if offset>1) or the access *r (if offset==1) would have undefined behavior.

A.2.3 Pointer equality comparison and provenance

This document reduces pointer equality tests to simple comparisons of the underlying abstract addresses, thus in general to numeric comparisons. For the following example, assume that x and y have consecutive abstract addresses.

// provenance_equality_global_xy.c

```
#include <stdio.h>
#include <string.h>
int x=1, y=2;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  _Bool b = (p==q);
  // can this be false even with identical addresses?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

Some existing implementations (including for example GCC 8.1 with -O2 optimization) sometimes regards two pointers with the same address but different provenance as non-equal. This happens in some circumstances but not others, e.g. if the test is pulled into a simple separate function, but not if in a separate compilation unit.

So within the proposed model, pointer equality alone is not suitable to determine that two pointers have different provenance. To the contrary, two pointers that have the same abstract address necessarily compare equal even if they have different provenance. Thus, implementations as described above are not conforming to this document.

A.3 Tracking provenance through integers

A different model of provenance also tracks provenance through integers and integer operations. It was motivated by observing the behavior and documentation of some implementations for **uintptr_t** analogues of the first test of A.2.

PVI: a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/pointer casts, and making some particular choices for the provenance results of integer and pointer +/- integer operations.

This model is only used here for the examples and in Annex B for semantics to clarify the differences to the models introduced in clause 4.2.1.

A.4 Operations on pointer values and representations

A.4.1 Pointer/integer casts

ISO/IEC 9899:2018 (6.3.2.3) leaves conversions between pointer and integer types implementation-defined, except for conversion of integer constant expressions of value 0 (which are null pointer constants) and for the optional **intptr_t** and **uintptr_t** types, for which it guarantees that any valid pointer to **void** may be converted and back, and that the result will compare equal to the original pointer. The PNVI-ae-udi model specified in by this document supports the assignment and lvalue conversion of *q for the following example by reconstructing the original provenance (the other variants of PNVI-* and PVI introduced above also support this example).

// provenance_roundtrip_via_intptr_t.c

```
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

A.4.2 Inter-object integer arithmetic

Below is a **uintptr_t** analogue of the example from clause A.2.2, attempting to move between objects with **uintptr_t** arithmetic. In PNVI-*, this has defined behavior. For PNVI-plain: the integer values are pure integers, and at the **int*** cast the value of ux+offset matches the address of y (live and of the right type), so the resulting pointer value takes on the provenance of the y storage instance. For PNVI-ae and PNVI-ae-udi, the storage instance for y is marked as *exposed* at the cast of &y to an integer, and so the above is likewise permitted there.

// pointer_offset_from_int_subtraction_global_xy.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = uy - ux;
  printf("Addresses: &x=%"PRIuPTR" &y=%"PRIuPTR"\
         " offset=%"PRIuPTR" \n",ux,uy,offset);
  int *p = (int *)(ux + offset);
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

In PVI, this has undefined behavior. First, the integer values of ux and uy have the provenances of the storage instances of x and y respectively. Then offset is a subtraction of two integer values with non-equal single provenances; the result of such an operation is defined to have the empty provenance. Adding that empty-provenance result to ux preserves the original x-storage instance provenance of the latter, as does the cast to **int***. Then the final *p=11 access is via a pointer value whose address is not consistent with its provenance.

Similarly, in the following PNVI-* allows (contrary to current GCC/ICC O2) a **uintptr_t** analogue of the first test of A.2. PVI forbids this test.

// provenance_basic_using_uintptr_t_global_xy.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = 4;
  ux = ux + offset;
  int *p = (int *)ux; // does this have UB?
  int *q = &y;
  printf("Addresses: &x=%p p=%p &y=%" PRIxPTR "\n", (void*)&x, (void*)p, uy);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // does this have undefined behaviour?
    printf("x=%d  y=%d  *p=%d  *q=%d\n", x, y, *p, *q);
  }
}
```

Thus PVI permits more aggressive alias analysis for pointers computed via integers (though those may be relatively uncommon), while PNVI-* will allow not just this test, which as written is probably not idiomatic desirable C, but also the essentially identical XOR doubly linked list idiom, using only one pointer per node by storing the XOR of two, as in the following.

// pointer_offset_xor_global.c

```
#include <stdio.h>
#include <inttypes.h>
int x=1;
int y=2;
int main() {
  int *p = &x;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  uintptr_t k = i ^ j;
  uintptr_t l = k ^ i;
  int *r = (int *)l;
  // are r and q now equivalent?
  *r = 11;     // does this have defined behaviour?
  _Bool b = (r==q);
  printf("x=%i y=%i *r=%i (r==p)=%s\n", x, y, *r, (b ? "true" : "false"));
}
```

There are other real-world but rare cases of inter-object arithmetic, e.g. in the implementations of Linux and FreeBSD per-CPU variables, in fixing up pointers after a **realloc**, and in dynamic linking (though arguably some of these are not between C abstract-machine objects). These are rare enough that it seems reasonable to require additional source annotation, or some other mechanism, to prevent compilers implicitly assuming that uses of such pointers as undefined.

A.4.3 Pointer provenance for pointer bit manipulations

It is a common idiom in systems code to use otherwise unused bits of pointers: low-order bits for pointers known to be aligned, and/or high-order bits beyond the addressable range. The example below assumes

- that **_Alignof(int)**>= 4,
- that the cast of a pointer to **uintptr_t** effectively has the two low-order bits of the result as 0, and
- that low-order bits are left untouched by casts in both directions between pointers and integers.

It casts a pointer to **uintptr_t** and back, using bitwise logical operations on the integer value to store a tag bit and to mask it out before the back cast to the pointer.

Under the above assumptions, in PNVI-* the intermediate value of q will have empty provenance, but the value of r used for the access will re-acquire the correct provenance at cast time. In PVI the binary operations used here (combining an integer value that has some provenance ID with a pure integer) preserve that provenance.

// provenance_tag_bits_via_uintptr_t_1.c

```
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p = &x;
  // cast &x to an integer
  uintptr_t i = (uintptr_t) p;
  // set low-order bit
  i = i | 1u;
  // cast back to a pointer
  int *q = (int *) i; // does this have UB?
  // cast to integer and mask out low-order bits
  uintptr_t j = ((uintptr_t)q) & ~((uintptr_t)3u);
  // cast back to a pointer
  int *r = (int *) j;
  // are r and p now equivalent?
  *r = 11;           //  does this have UB?
  _Bool b = (r==p);  //  is this true?
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"t":"f");
}
```

Note that in ISO/IEC 9899:2018 the result of the cast (**int***)i is implementation-defined, per 6.3.2.3p{5,6} and 7.20.1.4, and only works here because of the strong assumptions made above. An analogous example that only uses **uintptr_t** to store tagged pointer values and always mask such integer values before back conversion

```
  uintptr_t j = i & ~((uintptr_t)3u);
```

is guaranteed to work with the model specified in this document even without the third assumption in the list above.

A.4.4 Algebraic properties of integer operations

The PVI definitions of the provenance results of integer operations, chosen to make example A.4.2 forbidden and example A.4.3 allowed, have an unfortunate consequence: they make those operations no longer associative. Compare the examples below:

// pointer_arith_algebraic_properties_2_global.c

```
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(((uintptr_t)&(x[0])) +
    (((uintptr_t)&(y[1]))-((uintptr_t)&(y[0]))));
  *p = 11;  // is this free of undefined behaviour?
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

// pointer_arith_algebraic_properties_3_global.c

```
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(
    (((uintptr_t)&(x[0])) + ((uintptr_t)&(y[1])))
```



```
    -((uintptr_t)&(y[0])) );
  *p = 11;  // is this free of undefined behaviour?
  //(equivalent to the &x[0]+(&(y[1])-&(y[0])) version?)
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

The latter has undefined behavior in PVI. The PNVI-ae-udi model specified by this document and the other PNVI-* models do not suffer from this problem.

A.4.5 Copying pointer values with memcpy

The presented model allows this and makes the results usable for accessing memory, since **memcpy** and similar functions have to preserve the original provenance.

// pointer_copy_memcpy.c

```
#include <stdio.h>
#include <string.h>
int x=1;
int main() {
  int *p = &x;
  int *q;
  memcpy (&q, &p, sizeof p);
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

A.4.6 Copying pointer values bytewise, with user-memcpy

One of the key aspects of C is that it supports manipulation of object representations, e.g. as in the following naive user implementation of a **memcpy**-like function, which constructs a pointer value from copied bytes.

// pointer_copy_user_dataflow_direct_bytewise.c

```
#include <stdio.h>
#include <string.h>
int  x=1;
void user_memcpy(unsigned char* dest,
                 unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = *src;
    src += 1; dest += 1; n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy((unsigned char*)&q,
              (unsigned char*)&p, sizeof(int *));
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

The presented PNVI-plain model makes this valid: the representation bytes have no provenance, but when reading a pointer value from the copied memory, the read will be from multiple representation-byte writes. The same semantics for such reads are the same as for integer-to-pointer casts: checking at read-time that the address is within a live object, and giving the result the corresponding provenance.

PNVI-ae and PNVI-ae-udi, as specified by this document, mark storage instances as exposed whenever representation bytes of pointers to them are read, and use the same semantics for reads of pointer values from representation-byte writes as for integer-to-pointer casts. This means that integer-to-pointer casts become permitted for all storage instances for which a pointer has been copied via user_memcpy.

PVI makes user-**memcpy** legal by regarding each byte (as an integer value) as having the provenance of the original pointer, and the result pointer, being composed of representation bytes of which at least one has that provenance and none have a conflicting provenance, as having the same.

A.4.7 Pointer provenance for bytewise pointer representation manipulations

To examine the possible semantics for pointer representation bytes more closely, especially for PNVI-ae and PNVI-ae-udi, consider the following.

// provenance_tag_bits_via_repr_byte_1.c

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p=&x, *q=&x;
  // read low-order (little endian) representation byte of p
  unsigned char i = *(unsigned char*)&p;
  // check the bottom two bits of an int* are not used
  assert(_Alignof(int) >= 4);
  assert((i & 3u) == 0u);
  // set the low-order bit of the byte
  i = i | 1u;
  // write the representation byte back
  *(unsigned char*)&p = i;
  // [p might be passed around or copied here]
  // clear the low-order bits again
  *(unsigned char*)&p = (*(unsigned char*)&p) & ~((unsigned char)3u);
  // are p and q now equivalent?
  *p = 11;          // does this have defined behaviour?
  _Bool b = (p==q); // is this true?
  printf("x=%i *p=%i (p==q)=%s\n",x,*p,b?"true":"false");
}
```

As in A.4.3, this manipulates the low-order bits of a pointer value, but now it does so by manipulating one of its representation bytes instead of by casting to **uintptr_t** and back. In PNVI-plain and PVI this will just work, respectively reconstructing the original provenance and tracking it through the (changed and unchanged) integer bytes.

In PNVI-ae and PNVI-ae-udi, the storage instance of x is regarded as having been exposed by the read of the low-order representation byte of the pointer object p (with non-empty provenance in its abstract bytes in memory). Then the last reads of the value of p, from a combination of the original p=&x write and later integer byte writes, use the same semantics as integer-to-pointer casts, and thus recreate the original provenance.

A.4.8 Copying pointer values via control flow

Consider the exotic versions of **memcpy** in the two following examples that make a control-flow choice on the value of each bit or each byte, reconstructing each with constants in each control-flow branch.

// pointer_copy_user_ctrlflow_bytewise_abbrev.c

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <limits.h>
int  x=1;
unsigned char control_flow_copy(unsigned char c) {
  assert(UCHAR_MAX==255);
  switch (c) {
  case 0: return(0);
  case 1: return(1);
  case 2: return(2);
  ...
  case 255: return(255);
  }
}
void user_memcpy2(unsigned char* dest, unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = control_flow_copy(*src);
```



```
    src += 1;
    dest += 1;
    n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy2((unsigned char*)&q, (unsigned char*)&p, sizeof(p));
  *q = 11; // does this have undefined behaviour?
  printf("*p=%d  *q=%d\n", *p, *q);
}
```

// pointer_copy_user_ctrlflow_bitwise.c

```
#include <stdio.h>
#include <inttypes.h>
#include <limits.h>
int x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t)p;
  int uintptr_t_width = sizeof(uintptr_t) * CHAR_BIT;
  uintptr_t bit, j;
  int k;
  j=0;
  for (k=0; k<uintptr_t_width; k++) {
    bit = (i & (((uintptr_t)1) << k)) >> k;
    if (bit == 1)
      j = j | ((uintptr_t)1 << k);
    else
      j = j;
  }
  int *q = (int *)j;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

In all PNVI variants both examples are valid. In particular, for the model proposed in this document the first exposes the storage instance of the copied pointer value by representation-byte reads and the second by a pointer-to-integer cast.

For PVI, both examples have empty-provenance pointer values and hence the behavior is undefined.

A.4.9 Pointer provenance and union type punning

Pointer values can also be constructed in C by type punning, e.g. writing a pointer-type union member, reading it as a **uintptr_t** union member, and then casting back to a pointer type. The following example assumes that the object representation of the pointer and the object representation of the result of the cast to integer are identical. This property is not guaranteed by ISO/IEC 9899:2018, but holds for many implementations.

// provenance_union_punning_3_global.c

```
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
int x=1;
typedef union { uintptr_t ui; int *up; } un;
int main() {
  un u;
  int *p = &x;
  u.up = p;
  uintptr_t i = u.ui;
  int *q = (int*)i;
  *q = 11;  // does this have UB?
  printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  return 0;
}
```

For PNVI-ae and PNVI-ae-udi, the same semantics as for representation-byte reads also makes this case valid: the storage instance is exposed by the read of the representation bytes by the non-pointer-type read. The integer-to-pointer cast then recreates the provenance of x.

A.4.10 Pointer provenance via IO

Consider now pointer provenance flowing via IO, e.g. writing the address of an object to a string or stream and reading it back in. There are three possibilities: one using **fprintf**/**fscanf** and the %p format, one using **fwrite**/**fread** on the pointer representation bytes, and one converting the pointer to and from **uintptr_t** and using **fprintf**/**fscanf** on that value with the **PRIuPTR**/**SCNuPTR** formats. The following examples correspond to these three cases.

// provenance_via_io_percentp_global.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  FILE *f = fopen("provenance_via_io_percentp_global.tmp","w+b");
  printf("Addresses: p=%p\n", (void*)p);
  // print pointer address to a file
  fprintf(f, "%p\n", (void*)p);
  rewind(f);
  void *rv;
  int n = fscanf(f, "%p\n", &rv);
  int *r = (int *)rv;
  if (n != 1) exit(EXIT_FAILURE);
  printf("Addresses: r=%p\n", (void*)r);
  // are r and p now equivalent?
  *r=12;                                  // is this free of undefined behaviour?
  _Bool b1 = (r==p);                      // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r))); // same representations?
  printf("x=%i *r=%i b1=%s b2=%s\n", x, *r, b1 ? "true" : "false", b2 ? "true" : "false")
      ;
}
```

// provenance_via_io_bytewise_global.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  FILE *f = fopen("provenance_via_io_bytewise_global.tmp", "w+b");
  printf("Addresses: p=%p\n", (void*)p);
  // output pointer address to a file
  int nw = fwrite(&p, 1, sizeof(int *), f);
  if (nw != sizeof(int *)) exit(EXIT_FAILURE);
  rewind(f);
  int *r;
  int nr = fread(&r, 1, sizeof(int *), f);
  if (nr != sizeof(int *)) exit(EXIT_FAILURE);
  printf("Addresses: r=%p\n", (void*)r);
  // are r and p now equivalent?
  *r=12;                                  // is this free of undefined behaviour?
  _Bool b1 = (r==p);                      // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r))); //same representations?
  printf("x=%i *r=%i b1=%s b2=%s\n", x, *r, b1 ? "true" : "false", b2 ? "true" : "false")
      ;
}
```


// provenance_via_io_uintptr_t_global.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t)p;
  FILE *f = fopen("provenance_via_io_uintptr_t_global.tmp", "w+b");
  printf("Addresses: i=%"PRIuPTR" \n", i);
  // print pointer address to a file
  fprintf(f, "%" PRIuPTR "\n", i);
  rewind(f);
  uintptr_t k;
  // read a pointer address from the file
  int n = fscanf(f, "%" SCNuPTR "\n",&k);
  if (n != 1) exit(EXIT_FAILURE);
  printf("Addresses: k=%" PRIuPTR "\n",k);
  int *r = (int *)k;
  // are r and q now equivalent?
  *r=12;                                  // is this free of undefined behaviour?
  _Bool b1 = (r==p);                      // do they compare equal?
  _Bool b2 = (0==memcmp(&r,&p,sizeof(r))); //same representations?
  printf("x=%i *r=%i b1=%s b2=%s\n", x, *r, b1 ? "true" : "false", b2 ? "true" : "false")
      ;
}
```

Somewhat exotic though they are, these idioms are used in practice: in graphics code for serialization/deserialization (using %p), in xlib (using SCNuPTR), and in debuggers.

For fprintf and fscanf with the %p specifier, ISO/IEC 9899:2018 clearly permits this and the pointer value that is read is the same as the original one.

For the provenance in PNVI-ae and PNVI-ae-udi, the associated storage instance is exposed by the output, and the same semantics as for integer-to-pointer casts is used to attribute the same provenance to the input.

If the pointer value transits through uintptr_t or through representation-bytes, the storage instance are exposed the same.

A.5 Implications of provenance semantics for optimizations

In an ideal world, a memory object semantics for C would be consistent with all existing mainstream code usage and compiler behavior. In practice, it appears that these have diverged, making some compromise required. As already stated above, the PNVI-* semantics make some currently observed behavior on some implementations unsound, though at least some already regard that behavior as an unfixed bug, due to the lack of integer/pointer type distinctions in the internal representation. The following sections consider some other important cases, by example.

A.5.1 Optimization based on equality tests

All PNVI-* and PVI models let p==q hold for some pointer values p and q in cases where interchanging them would in fact have undefined behavior. In the context of the LLVM intermediate representation[7] this can limit optimizations such as global value numbering based on pointer equality tests. PVI suffers from this problem also for integer comparisons (hence much more severely), wherever the integers might have provenance information and are then eventually cast back to a pointer.

A.5.2 Can a function parameter alias local variables of the function?

An access through a function parameter to a local variable inhibits a number of optimizations, therefore one goal of this document is to mark such an access as undefined behavior. Consider first the example below,

where **main**() guesses the address of f()'s local variable, passing it in as a pointer, and f() checks it before using it for an access. Here some implementations optimize away the **if** and the store operation *p=7, even in executions where the ADDRESS_PFI_1PG constant is the same as the abstract address of j. A goal of the specification in this document is to make such an optimization valid, so, in a situation where equality holds, it deems this program to have undefined behavior. In other words, programs may not rely on implementation facts about the allocation addresses of C variables.

// pointer_from_integer_1pg.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1PG;
  int *p = (int*)i;
  f(p);
}
```

In the PNVI-* semantics as proposed by this document the behavior is undefined because at the point of the (**int***)i cast the j storage instance does not yet exist, so the cast gives a pointer with empty provenance; any execution that goes into the **if** would thus be undefined. The PVI semantics has undefined behavior for the simple reason that j is created with the empty provenance, and hence p inherits that.

Varying to do the cast to **int*** in f() instead of **main**(), passing in an integer i instead of a pointer, this becomes defined in PNVI-plain, as j exists at the point of the (**int***)i cast. But in PNVI-ae and PNVI-ae-udi, the storage instance of j is not exposed, so the cast to **int*** gives a pointer with empty provenance. The access via this pointer is thus undefined and optimizations as indicated above are thus valid.

// pointer_from_integer_1ig.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1IG;
  f(j);
}
```

In PVI, this example also has undefined behavior.

A.5.3 Variants of integer to pointer conversions

Note that both of the previous examples take the address of j to guard their *p=7 accesses. Removing the conditional guards gives the two tests below, for which this document aims that that behavior is undefined if u holds the abstract address of j:

// pointer_from_integer_1p.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  *p=7;
  printf("j=%d\n",j);
```



```
}
int main() {
  uintptr_t u = ADDRESS_PFI_1P;
  int *p = (int*)u;
  f(p);
}
```

// pointer_from_integer_1i.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t u = ADDRESS_PFI_1I;
  f(u);
}
```

Both are undefined in PVI for the same reason as before, and the first is forbidden in PNVI-*, again because j does not exist at the cast point. The second is undefined as intended for PNVI-ae and PNVI-ae-udi because the storage instance of j is never exposed.

To change the second example such that it is valid when u holds the abstract address of j the storage instance has to be exposed, for example by taking the address:

// pointer_from_integer_1ie.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t u = ADDRESS_PFI_1I;
  f(u);
}
```

A.5.4 Access local variables of a parent function call

Similar to the above, access to a local variable from a called function inhibits a number of optimizations, therefore one goal of this document is to mark such an access as undefined behavior. In executions where the ADDRESS_PFI_2 constant is the abstract address of j, in PVI the example below again has undefined behavior for the simple reason that p has the empty provenance. The same holds for PNVI-ae and PNVI-ae-udi, because j is never exposed.

// pointer_from_integer_2.c

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
```



```
11    f();
12    printf("j=%d\n",j);
13  }
```

In contrast to that, in PNVI-ae and PNVI-ae-udi the following program has defined behavior, because the `j` storage instance is exposed by the `(uinptr_t)&j` cast.

// pointer_from_integer_2g.c

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include "charon_address_guesses.h"
4  void f() {
5    uintptr_t i=ADDRESS_PFI_2G;
6    int *p = (int*)i;
7    *p=7;
8  }
9  int main() {
10   int j=5;
11   if ((uintptr_t)&j == ADDRESS_PFI_2G)
12     f();
13   printf("j=%d &j=%p\n",j,(void*)&j);
14 }
```

This does not match the behavior of some implementations that do not yet follow the specification of this document. For example, current Clang at optimization levels O2 or O3, assumes that `j` does not alias and prints `j=5` unconditionally.

A.5.5 Possible loss of exposure information between translation paths

The PVNI-ae-udi model specified by this document has the property of identifying undefined behaviors without requiring examination of multiple executions. It has the disadvantage that whether a storage instance has been exposed is a fragile syntactic property, and is e.g. not preserved by dead code elimination. Here, examples like the one below [Richard Smith, personal communication], in which the code correctly guesses the abstract address of a storage instance and adds that to a zero-valued quantity have defined behavior.

// provenance_lost_escape_1.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdint.h>
4  #include "charon_address_guesses.h"
5  int x=1; // assume allocation ID @1, at ADDR_PLE_1
6  int main() {
7    int *p = &x;
8    uintptr_t i1 = (intptr_t)p;            // (@1,ADDR_PLE_1)
9    uintptr_t i2 = i1 & 0x00000000FFFFFFFF;//
10   uintptr_t i3 = i2 & 0xFFFFFFFF00000000;// (@1,0x0)
11   uintptr_t i4 = i3 + ADDR_PLE_1;        // (@1,ADDR_PLE_1)
12   int *q = (int *)i4;
13   printf("Addresses: p=%p\n",(void*)p);
14   if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
15     *q = 11;  // does this have defined behaviour?
16     printf("x=%d *p=%d *q=%d\n",x,*p,*q);
17   }
18 }
```

This code is valid for the model proposed here, because the address of `x` is both taken and cast to an integer type. However, implementations may perform algebraic optimizations before alias analysis, and those optimizations may erase the `&x` operation and subsequent cast, replacing it and all the calculation of `i3` by zero. But then alias analysis is not unable to see that `x` has been exposed and thus that it is valid that `*q` accesses `x`.

If these semantics were used for alias analysis in an intermediate language after such optimization, this would require the optimization passes to record which addresses have been taken and cast to integer (or otherwise exposed) in eliminated code, to be explicitly passed in to alias analysis.

A.5.6 One-past integer-to-pointer casts

PNVI-ae-udi is designed to permit a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer semantic check that *q is properly within the footprint of the storage instance by a check that it is properly within or one-past.

// provenance_roundtrip_via_intptr_t_onepast.c

```
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  p=p+1;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  q=q-1;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

The PNVI-ae-udi approach specified in this document is to leave the provenance of pointer values resulting from such casts ambiguous (see 4.2.6) until the first operation (e.g. an access, pointer arithmetic, or pointer relational comparison) that disambiguates them. This makes the following two examples, each of which uses the result of the cast in one consistent way, well defined:

// pointer_from_int_disambiguation_1.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0)
      {
    int *r = (int *)i;
    *r=11;  // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d *r
        =%d\n",x,y,*p,*q,*r);
  }
}
```

// pointer_from_int_disambiguation_2.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0)
      {
    int *r = (int *)i;
    r=r-1;  // is this free of UB?
    *r=11;  // and this?
    printf("x=%d y=%d *p=%d *q=%d *r
        =%d\n",x,y,*p,*q,*r);
  }
}
```

while making the behavior of the following, which tries to use the result of the cast to access both objects, undefined.

// pointer_from_int_disambiguation_3.c

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    int *r = (int *)i;
    *r=11;
    r=r-1;  // is this free of UB?
    *r=12;  // and this?
```

```
16      printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y,*p,*q,*r);
17    }
18  }
```

In this, the assignment `*r=11` will first disambiguate the provenance of the pointer value in one way (choosing the storage instance of `y`). Then the pointer arithmetic of `r-1` steps out-of-bounds before the address range of that storage instance and the operation thus has undefined behavior.

A.6 Testing the example behavior in Cerberus

Executable versions of the PNVI-plain, PNVI-ae, and PNVI-ae-udi models have been implemented in Cerberus[9, 10, 8], closely following the detailed semantics of Annex B. This makes it possible to interactively or exhaustively explore the behavior of the examples, confirming that they are allowed or not as intended.

		intended behaviour			observed behaviour Cerberus (decreasing allocator)		
test family	test	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi
1	provenance_basic_global_xy.c	UB			not triggered		
	provenance_basic_global_yx.c				UB (line 9)		
	provenance_basic_auto_xy.c				not triggered		
	provenance_basic_auto_yx.c				UB (line 9)		
2	**cheri_03_ii.c**	UB			UB (except with *permissive_pointer_arith* switch)		
3	**pointer_offset_from_ptr_subtraction_global_xy.c**	UB (pointer subtraction)			UB (pointer subtraction) Or UB (out-of-bound store with *permissive_pointer_arith* switch)		
	pointer_offset_from_ptr_subtraction_global_yx.c						
	pointer_offset_from_ptr_subtraction_auto_xy.c						
	pointer_offset_from_ptr_subtraction_auto_yx.c						
4	**provenance_equality_global_xy.c**	defined, nondet			not triggered		
	provenance_equality_global_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_auto_xy.c				not triggered		
	provenance_equality_auto_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_global_fn_xy.c				not triggered		
	provenance_equality_global_fn_yx.c				defined (ND except with *strict pointer equality* switch)		
5	**provenance_roundtrip_via_intptr_t.c**	defined			defined		
6	**provenance_basic_using_uintptr_t_global_xy.c**	defined			not triggered		
	provenance_basic_using_uintptr_t_global_yx.c				defined		
	provenance_basic_using_uintptr_t_auto_xy.c				not triggered		
	provenance_basic_using_uintptr_t_auto_yx.c				defined		
7	**pointer_offset_from_int_subtraction_global_xy.c**	defined			defined		
	pointer_offset_from_int_subtraction_global_yx.c				defined		
	pointer_offset_from_int_subtraction_auto_xy.c				defined		
	pointer_offset_from_int_subtraction_auto_yx.c				defined		
8	**pointer_offset_xor_global.c**	defined			defined		
	pointer_offset_xor_auto.c				defined		
9	provenance_tag_bits_via_uintptr_t_1.c	defined			defined		
10	**pointer_arith_algebraic_properties_2_global.c**	defined			defined		
11	**pointer_arith_algebraic_properties_3_global.c**	defined			defined		
12	**pointer_copy_memcpy.c**	defined			defined		
13	**pointer_copy_user_dataflow_direct_bytewise.c**	defined			defined		
13	**provenance_tag_bits_via_repr_byte_1.c**	defined			defined		
15	pointer_copy_user_ctrlflow_bytewise.c	defined			defined		
16	**pointer_copy_user_ctrlflow_bitwise.c**	defined			defined		
17	provenance_equality_uintptr_t_global_xy.c	defined			not triggered		
	provenance_equality_uintptr_t_global_yx.c				defined (true)		
	provenance_equality_uintptr_t_auto_xy.c				not triggered		
	provenance_equality_uintptr_t_auto_yx.c				defined (true)		
18	provenance_union_punning_2_global_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_global_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
	provenance_union_punning_2_auto_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_auto_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
19	**provenance_union_punning_3_global.c**	defined			defined		
20	provenance_via_io_percentp_global.c	**filesystem and scanf() are not currently supported by Cerberus**					
	provenance_via_io_bytewise_global.c						
	provenance_via_io_uintptr_t_global.c						
21	**pointer_from_integer_1pg.c**	UB (line 7)			UB in one exec (line 7)		
	pointer_from_integer_1ig.c	defined (j = 7)	UB (line 8)		defined (j = 7)	UB (line 8)	
	pointer_from_integer_1p.c	UB (line 6)			UB (line 6)		
	pointer_from_integer_1i.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_1ie.c	defined (j = 7)			defined (j = 7)		
	pointer_from_integer_2.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_2g.c	defined (j = 7)			defined (j = 7)		
	provenance_lost_escape_1.c	defined			defined		
22	**provenance_roundtrip_via_intptr_t_onepast.c**	UB (line 10)		defined	UB (line 10)		defined
23	**pointer_from_int_disambiguation_1.c**	defined (y = 11)			defined (y = 11)		
	pointer_from_int_disambiguation_1_xy.c				not triggered		
	pointer_from_int_disambiguation_2.c	UB (line 14)		defined	UB (line 14)		defined (x = 11)
	pointer_from_int_disambiguation_2_xy.c				not triggered		
	pointer_from_int_disambiguation_3.c	UB (line 15)		UB (line 15)	UB (line 15)		
	pointer_from_int_disambiguation_3_xy.c				not triggered		

(**bold** = tests mentioned in the document)

green = Cerberus behaviour matches intent
blue = Cerberus behaviour matches intent (witch *permissive_pointer_arith* switch)
grey = Cerberus' allocator doesn't trigger the interesting behaviour

A.7 Testing the example behavior in mainstream C implementations

Some tests rely on address coincidences for the interesting execution; for these sometimes multiple variants were included, tuned to the allocation behavior in the implementations that were considered. Where this has not been done, some of the experimental data is not meaningful.

The detailed data is available via the Cerberus tool[9], and summarized in the table below.

Compilers

test family	test	Observed behaviour (compilers), sound w.r.t PNVI-*? (relying on UB or ND?)								
		gcc-8.3			clang-7.0.1			icc-19		
		PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi
1	provenance_basic_global_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_global_yx.c		y (y for O2+)			not triggered			not triggered	
	provenance_basic_auto_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_auto_yx.c		y (n)			y (n)			y (y for O2+)	
2	**cheri_03_ii.c**		y (n)			y (n)			y (n)	
3	**pointer_offset_from_ptr_subtraction_global_xy.c**								y (n)	
	pointer_offset_from_ptr_subtraction_global_yx.c								y (n)	
	pointer_offset_from_ptr_subtraction_auto_xy.c		y (n)			y (n)			y (y for O2+)	
	pointer_offset_from_ptr_subtraction_auto_yx.c								y (y for O2+)	
4	**provenance_equality_global_xy.c**		y (n)							
	provenance_equality_global_yx.c		y (y for O2+)							
	provenance_equality_auto_xy.c		y (y for O2+)							
	provenance_equality_auto_yx.c		y (n)			y (n)			y (n)	
	provenance_equality_global_fn_xy.c		y (n)							
	provenance_equality_global_fn_yx.c		y (y for O2+)							
5	**provenance_roundtrip_via_intptr_t.c**		y (n)			y (n)			y (n)	
6	**provenance_basic_using_uintptr_t_global_xy.c**		y (n)			y (n)			**n (y)**	
	provenance_basic_using_uintptr_t_global_yx.c		**n (y)**			not triggered			not triggered	
	provenance_basic_using_uintptr_t_auto_xy.c		y (n)			not triggered			**n (y)**	
	provenance_basic_using_uintptr_t_auto_yx.c		y (n)			y (n)			**n (y)**	
7	**pointer_offset_from_int_subtraction_global_xy.c**									
	pointer_offset_from_int_subtraction_global_yx.c									
	pointer_offset_from_int_subtraction_auto_xy.c		y (n)			y (n)			y (n)	
	pointer_offset_from_int_subtraction_auto_yx.c									
8	**pointer_offset_xor_global.c**									
	pointer_offset_xor_auto.c		y (n)			y (n)			y (n)	
9	provenance_tag_bits_via_uintptr_t_1.c		y (n)			y (n)			y (n)	
10	**pointer_arith_algebraic_properties_2_global.c**		y (n)			y (n)			y (n)	
11	**pointer_arith_algebraic_properties_3_global.c**		y (n)			y (n)			y (n)	
12	**pointer_copy_memcpy.c**		y (n)			y (n)			y (n)	
13	**pointer_copy_user_dataflow_direct_bytewise.c**		y (n)			y (n)			y (n)	
13	**provenance_tag_bits_via_repr_byte_1.c**		y (n)			y (n)			y (n)	
15	pointer_copy_user_ctrlflow_bytewise.c		y (n)			y (n)			y (n)	
16	**pointer_copy_user_ctrlflow_bitwise.c**		y (n)			y (n)			y (n)	
17	provenance_equality_uintptr_t_global_xy.c									
	provenance_equality_uintptr_t_global_yx.c									
	provenance_equality_uintptr_t_auto_xy.c		y (n)			y (n)			y (n)	
	provenance_equality_uintptr_t_auto_yx.c									
18	provenance_union_punning_2_global_xy.c		y (n)			y (n)		**n (y)**	y (y for O2+)	
	provenance_union_punning_2_global_yx.c	**n (y)**	y (y for O2+)			not triggered			not triggered	
	provenance_union_punning_2_auto_xy.c		y (n)			y (n)		**n (y)**	y (y for O2+)	
	provenance_union_punning_2_auto_yx.c		y (n)					**n (y)**	y (y for O2+)	
19	**provenance_union_punning_3_global.c**		y (n)			y (n)			y (n)	
20	provenance_via_io_percentp_global.c									
	provenance_via_io_bytewise_global.c		NO OPT			NO OPT			NO OPT	
	provenance_via_io_uintptr_t_global.c									
21	**pointer_from_integer_1pg.c**		y (y for O0+)			y (y for O2+)			y (y for O2+)	
	pointer_from_integer_1ig.c	**n (y)**	y (y for O2+)		**n (y)**	y (y for O2+)		**n (y)**	y (y for O2+)	
	pointer_from_integer_1p.c									
	pointer_from_integer_1i.c									
	pointer_from_integer_1ie.c		**can't test with charon**							
	pointer_from_integer_2.c									
	pointer_from_integer_2g.c		y (n)			**n (y)**			y (n)	
	provenance_lost_escape_1.c		y (n)			y (n)			**n (y for O2+)**	
22	**provenance_roundtrip_via_intptr_t_onepast.c**		y (n)			y (n)			y (n)	
23	**pointer_from_int_disambiguation_1.c**		**n (y)**			not triggered			not triggered	
	pointer_from_int_disambiguation_1_xy.c		not triggered			y (n)			**n (y for O2+)**	
	pointer_from_int_disambiguation_2.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_2_xy.c		not triggered			y (n)			y (n)	
	pointer_from_int_disambiguation_3.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_3_xy.c		not triggered			y (n)			y (y for O2+)	

(**bold** = tests mentioned in the document)

Page 2

Annex B
(informative)

Detailed semantics

B.1 General

This annex gives detailed mathematical semantics for four variants of C provenance semantics **PNVI-plain**, **PNVI-ae**, **PNVI-ae-udi**, as specified in clause 4.2.1, and **PVI**, as specified in annex A.3. The term PNVI-* denotes the PNVI-plain, PNVI-ae, and PNVI-ae-udi.

The PNVI-ae and PNVI-ae-udi variants of PNVI permit bytewise copy of a pointer to an initially unexposed object, but leaves it marked as exposed.

B.2 The PNVI-ae-udi, PNVI-ae, PNVI-plain, and PVI semantics

These semantic definitions are manually typeset mathematics simplified from the executable-as-test-oracle Cerberus source (expressed in the pure-functional Lem definition language[11]). Most subobject details, function pointers, and some options, have been removed. Neither the typeset models or the Lem source consider linking, or pointers constructed via I/O (e.g. via %p or representation-byte I/O).

The memory object semantics is designed to be combined with a semantics for the thread-local semantics of the rest of C (expressed in Cerberus as a translation from C source to the *Core* intermediate language, together with an operational semantics for Core) to give a complete semantics for a large fragment of sequential C.

For simplicity, this assumes that pointer representations are the two's complement representation of their addresses (and identical to the two's complement representations of their conversions to sufficiently wide integer types), assume null pointers have address (and representation) 0, and allow null pointers to be constructed from any empty-provenance integer zero, not just integer constant expressions.

At present, the mathematical model does not include the semantics of ISO/IEC 9899:2018 that makes all pointers to an object or region invalid at the end of its lifetime, and it permits equality comparison between pointers irrespective of whether the objects of their provenances are live, but it permits pointer subtraction, relational comparison, array offset, member offset, and casts to integer only for pointers to live objects for which the address is within or one past the object footprint. These are all deliberate choices. One could instead check only that the addresses are within or one past the original object footprint (and not check the object is live), or go further towards a concrete-address view of pointer values and not check that either. Sketching out some of the options:

- zombie-pointers-become-indeterminate For the current semantics of ISO/IEC 9899:2018, at the end of every storage instance's lifetime any pointer value referring to it becomes invalid. After that, every memory footprint containing a pointer value with that provenance (result of the store of such a pointer value) has an indeterminate representation. With this, the live-object evaluations and checks for equality, relational comparison, subtraction, array offset member offset, and casts to integers all become moot.

- zombie-pointers-allow-equality-only This is what the maths below details.

- zombie-pointers-allow-all-in-bounds-arithmetic For this, one would retain metadata for the bounds of lifetime-ended pointers and check against that for non-load/store operations.

- zombie-pointers-allow-all-arithmetic For this, one would remove the lifetime and bounds checks for non-load/store operations.

- all-pointers-allow-all-arithmetic This would make all the non-load/store operations operate just on abstract addresses, ignoring provenance and storage instance metadata.

B.2.1 The memory object model interface

In Cerberus, the memory object model is factored out from Core with a clean interface[?]Fig. 2]Cerberus-PLDI16. This provides functions for memory operations:

- allocate_object (for objects with automatic or static storage duration, i.e. global and local variables),
- allocate_region (for the results of **malloc**, **calloc**, and **realloc**, i.e. heap-allocated regions),
- kill (for lifetime end of both kinds of allocation),
- load, and
- store,

and for pointer/integer operations: arithmetic, casts, comparisons, offseting pointers by struct-member offsets, etc. The interface involves types `pointer_value` (p), `integer_value` (x), `floating_value`, and `mem_value` (v), which are abstract as far as Core is concerned. Distinguishing pointer and integer values gives more precise internal types.

In PNVI-ae, PNVI, and PVI, a provenance π is either `@`i where i is a storage-instance ID, or the *empty* provenance `@empty`. In PNVI-ae-udi a provenance may also be a symbolic storage instance ID ι (iota), initially associated to two storage instance IDs and later resolved to one or the other.

A pointer value may either be `null` or a pair (π, a) of a provenance π and address a. In PNVI*, an integer value is simply a mathematical integer (within the appropriate range for the relevant C type), while in PVI, an integer value is a pair (π, n) of a provenance π and a mathematical integer n.

Memory values are the storable entities, either a pointer, integer, floating-point, array, struct, or union value, or `unspec` for unspecified values, each together with their C type.

B.3 The memory object model state

In both PVI and PNVI*, a memory state is a pair (A, M). The A is a partial map from storage-instance IDs to either `killed` or storage-instance metadata $(n, \tau_{\mathsf{opt}}, a, f, k, t)$:

- size n,
- optional C type τ (or `none` for allocated regions),
- base address a,
- permission flag $f \in \{\texttt{readWrite}, \texttt{readOnly}\}$,
- kind $k \in \{\texttt{object}, \texttt{region}\}$, and
- for PNVI-ae and PNVI-ae-udi, a taint flag $t \in \{\texttt{unexposed}, \texttt{exposed}\}$.

In PNVI-ae-udi, A also maps all symbolic storage instance IDs ι, to sets of either one or two (non-symbolic) storage instance IDs. One might also need to record a partial equivalence relation over symbolic storage instance IDs, to cope with the pointer subtraction and relational comparison cases where one learns that two provenances are equal but both remain ambiguous, but that is not spelt out in this document.

The M is a partial map from addresses to abstract bytes, which are triples of a provenance π, either a byte b or `unspec`, and an optional integer pointer-byte index j (or `none`). The last is used in PNVI* to distinguish between loads of pointer values that were written as whole pointer writes vs those that were written byte-wise or in some other way.

B.3.1 Mappings between abstract values and representation abstract-byte sequences

The M models the memory state in terms of low-level abstract bytes, but `store` and `load` take and return the higher-level memory values. The two are related with functions repr(v), mapping a memory value to a list of abstract bytes, and abst(τ, bs), mapping a list of abstract bytes bs to its interpretation as a memory value with C type τ.

The repr(v) function is defined by induction over the structure of its memory value parameter and returns a list of sizeof(τ) abstract bytes, where τ is the C type of the parameter. The base cases are values with scalar types (integer, floating and pointers) and unspecified values. For an unspecified value of type τ, it returns a list with abstract bytes of the form $(\texttt{@empty}, \mathsf{unspec}, \mathsf{none})$. Non-null pointer values are represented with lists of abstract bytes that each have the provenance of the pointer value, the appropriate part of the two's complement encoding of the address, and the $0..$ sizeof$(\tau) - 1$ index of each byte. Null pointers are represented with lists of abstract bytes of the form $(\texttt{@empty}, 0, \mathsf{none})$. In PVI, integer values are represented similarly to pointer values except that the third component of each abstract byte is none. In PNVI*, integer values are represented by lists of abstract bytes, with each of their first components always the empty provenance, and each of their third components again none. Floating-point values are similar, in all the models, except that the provenance of the abstract bytes is always empty. For array and struct/union values the function is inductively applied to each subvalue and the resulting byte-lists concatenated. The layout of structs and unions follow an implementation-defined ABI, with padding bytes like those of unspecified values.

The abst(τ, bs) function is defined by induction over τ. The base cases are again the scalar types. For these, sizeof(τ) abstract bytes are consumed from bs and a scalar memory value is constructed from their second components: if any abstract byte has an unspec value, an unspecified value is constructed; otherwise, depending on τ, a pointer, integer or floating-point value is constructed using the two's complement or floating-point encoding. For pointers with address 0, the provenance is empty. For non-0 pointer values and integer values, in PVI the provenance is constructed as follows: if at least one abstract byte has non-empty provenance and all others have either the same or empty provenance, that provenance is taken, otherwise the empty provenance is taken. In PNVI*, when constructing a pointer value, if the third components of the bytes all carry the appropriate index, and all have the same provenance (which will be guaranteed if pointer types all have the same size), the provenance of the result is that provenance. Otherwise, the A part of the memory state is examined to find whether a live storage instance exists with a footprint containing the pointer value that is being constructed. If so, in PNVI-plain, its storage instance ID is used for the provenance of the pointer value, otherwise the empty provenance is used.

- In PNVI-ae and PNVI-ae-udi, when constructing a pointer value, if A has to be examined then, matching the relevant integer-to-pointer cast semantics below, the storage instance has been exposed, otherwise the result have the empty provenance.
- In PNVI-ae-udi, if there are two such live storage instances, with IDs i_1 and i_2, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This may only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second.

For array/struct types, $abst()$ recurses on the progressively shrinking list of abstract bytes.

B.3.2 Memory operations

The successful semantics of memory operations is expressed as a transition relation between memory states, with transitions labeled by the operation (including its arguments) and return value:

$$(A, M) \xrightarrow{\text{label}} (A', M')$$

For example, the transitions

$$(A, M) \xrightarrow{\text{load}(\tau,p)=v} (A', M')$$

describe the semantics of a load(τ, p) in memory state (A, M), returning value v and with resulting memory state (A', M'). The semantics also defines when each operation flags an out-of-memory (OOM) or undefined behavior (UB) in a memory state (A, M).

B.3.2.1 Storage instance creation

When a new storage instance is created, either with allocate_region (for the results of **malloc**, **calloc**, and **realloc**, i.e. heap-allocated regions), or with allocate_object (for objects with automatic or static storage duration, i.e. global and local variables), in non-const and const variants: a fresh storage-instance ID i is chosen; an address a is chosen from newAlloc(A, al, n), defined to be the set of addresses of blocks of size n aligned by al that do not overlap with 0 or any other allocation in A; and the pointer value $p = (@i, a)$ is returned. In all three cases the storage-instance metadata A is updated with a new record for i. For PNVI-ae this is initially marked as unexposed. In the allocate_object case the size n of the allocation is the representation size of the C type τ. In the allocate_region$(al, \tau, \texttt{readOnly}(v))$ case, the last of the three rules, the memory M is updated to contain the representation of v at the addresses $a..a + \text{sizeof}(\tau) - 1$.

$$\frac{\text{[label: allocate\_region}(al, n) = p] \quad i \notin \text{dom}(A) \qquad a \in \text{newAlloc}(A, al, n) \qquad p = (@i, a)}{A, M \to A[i \mapsto (n, \texttt{none}, a, \texttt{readWrite}, \texttt{region}, \texttt{unexposed})], M}$$

$$\frac{\text{[label: allocate\_object}(al, \tau, \texttt{readWrite}) = p] \quad i \notin \text{dom}(A) \qquad a \in \text{newAlloc}(A, al, n) \qquad n = \text{sizeof}(\tau) \qquad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readWrite}, \texttt{object}, \texttt{unexposed})), M}$$

$$\frac{\text{[label: allocate\_object}(al, \tau, \texttt{readOnly}(v)) = p] \quad i \notin \text{dom}(A) \qquad a \in \text{newAlloc}(A, al, n) \qquad n = \text{sizeof}(\tau) \qquad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readOnly}, \texttt{object}, \texttt{unexposed})), M([a..a + n - 1] \mapsto \text{repr}(v))}$$

B.3.2.2 Storage instance lifetime end

When the storage instance of a pointer value $(@i, a)$ is killed, either by a `free()` for a heap-allocated region or at the end of lifetime of an object with automatic storage duration, the storage-instance metadata A of storage instance i is updated to record that i has been killed.

$$\frac{\text{[label: kill}(p, k)] \quad p = (@i, a) \qquad k = k' \qquad A(i) = (n, \_, a, f, k', \_)}{A, M \to A(i \mapsto \texttt{killed}), M}$$

B.3.2.3 Load

To load a value v of type τ from a pointer value $p = (@i, a)$, there is be a live storage instance for i in A, the footprint of τ at a is within the footprint of that allocation, and the value v is the abstract value obtained from the appropriate memory bytes from M.

$$\frac{\text{[label: load}(\tau, p) = v] \quad p = (@i, a) \qquad A(i) = (n, \_, a', \_, \_, \_) \qquad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1] \qquad v = \text{abst}(\tau, M[a..a + \text{sizeof}(\tau) - 1])}{A, M \to A, M}$$

For PNVI-ae and PNVI-ae-udi, if the recursive-on-τ computation of abst$(\tau, M[a..a + \text{sizeof}(\tau) - 1])$ involves a call of abst at any non-pointer scalar type for a region of M including an abstract byte with non-empty provenance, and the corresponding storage instance is live, it is marked as exposed. This applies e.g. for reads of pointer values via **char*** pointers, and for union type punning reads at **uintptr_t** of pointer values.

B.3.2.4 Store

To store a value v of type τ to a pointer value $p = (@i, a)$, there is be a live storage instance for i in A, which

is writable, and the footprint of τ at a is within the footprint of that allocation. The memory M is updated with the representation bytes of the value v.

$$\frac{\begin{array}{l}\text{[label: store}(\tau, p, v)\text{]} \\ p = (\text{@}i, a) \qquad A(i) = (n, \_, a', \texttt{readWrite}, \_, \_) \\ \qquad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1]\end{array}}{A, M \to A, M([a..a + \text{sizeof}(\tau) - 1] \mapsto \text{repr}(v))}$$

For PNVI-ae-udi, the kill, load, and store rules above are adapted. If $p = (\iota, a)$ and $A(\iota) = \{i\}$, the other premises and conclusion of the appropriate above rule apply. If $A(\iota) = \{i_1, i_2\}$ and the premises are satisfied for one of the two, say i_j, the rest of the rule applies except that in the final state A is additionally updated to map ι to $\{i_j\}$.

The memory operations flag out-of-memory (OOM) and undefined behavior (UB) as follows:

allocate_region(al, n) / allocate_object$(al, \tau, \texttt{readwrite})$ / allocate_object$(al, \tau, \texttt{readOnly}(v))$:	
OOM out of memory	if newAlloc$(A, al, n) = \{\}$ or newAlloc$(A, al, \text{sizeof}(\tau)) = \{\}$
load(τ, p) / store(τ, p, v) / kill(p):	
UB null pointer	if $p = \texttt{null}$
UB empty provenance	if $p = (\texttt{@empty}, a)$
UB killed provenance	if $p = (\text{@}i, a)$ and $A(i) = \texttt{killed}$
load(τ, p) / store(τ, p, v):	
UB out of bounds	if $p = (\text{@}i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $[a..a + \text{sizeof}(\tau) - 1] \not\subseteq [a'..a' + n - 1]$
store(τ, p, v):	
UB read-only	if $p = (\text{@}i, a)$ and $A(i) = (n, \_, a', \texttt{readOnly}, k, \_)$
kill(p):	
UB non-alloc-address	if $p = (\text{@}i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $a \neq a'$

For PNVI-ae-udi, the rules above are adapted. In the case where $p = (\iota, a)$ and $A(\iota) = \{i\}$, the semantics is exactly as for $p = (i, a)$, while if $A(\iota) = \{i_1, i_2\}$, one has UB only if the conditions above apply to both i_1 and i_2.

B.3.3 Pointer / Integer operations

B.3.3.1 Pointer subtraction

Pointers $p = (\text{@}i, a)$ and $p' = (\text{@}i', a')$ may be subtracted if they have the same provenance ($i = i'$), there is a live storage instance for i in A, and both a and a' are within or one-past the footprint of that allocation (in ISO/IEC 9899:2018 the last will always hold, otherwise UB would have been flagged in earlier pointer arithmetic). Otherwise UB. The result is the numerical difference $a - a'$ divided by sizeof(dearray(τ))), where dearray(τ) returns τ if it is not an array type, and otherwise returns its element type. Note that this disallows subtraction for which one or both arguments are null pointers, which is the semantics of ISO/IEC 9899:2018.

This rule is stated for PNVI and PNVI-ae, returning pure integer. For PVI, diff_ptrval constructs the same integer but with @empty provenance.

For PNVI-ae-udi, because subtraction of pointers with different provenance is UB:

- if both the two pointers have either a provenance @i (resp. @i') or a symbolic storage instance ID ι (resp. ι') mapped by A to a singleton $\{i\}$ (resp. $\{i'\}$), then $i = i'$, otherwise UB.
- if one of the two pointers has a symbolic storage instance ID ι, mapped by A to $\{i_1, i_2\}$, while the other either has a provenance @i' or an ι' mapped to a singleton $\{i'\}$, then i' is either i_1 or i_2, and ι is resolved to that in the A of the final state. Otherwise UB.
- If both pointers are ambiguous, say mapped to $\{i_1, i_2\}$ and $\{i'_1, i'_2\}$, then if those two sets share exactly one element which satisfies the other rule preconditions, both symbolic storage instance IDs are resolved to that. Otherwise UB.
- If both pointers are ambiguous and those sets share two elements that satisfy the other conditions (which is believed to only happen if the addresses are equal), then subtraction is permitted but the symbolic storage instance IDs are left unresolved. Otherwise UB.

For example, suppose p and q have been produced by separate casts from an integer which is ambiguously one-past one allocation and at the start of another. Then after p-q or p<q they have the same provenance, but any of the two is possible. (Alternatively, one could change the semantics to record an identity relation over symbolic storage instance IDs, and additional modifications to the rules below beyond what is in this document, but that seems to be unwarranted complexity).

$$\frac{\begin{array}{l}\text{[label: diff\_ptrval}(\tau, p, p') = x\text{]} \\ p = (\text{@}i, a) \qquad p' = (\text{@}i', a') \qquad i = i' \qquad A(i) = (n, \_, \hat{a}, \_, \_, \_) \\ x = (a - a')/\ \text{sizeof}(\text{dearray}(\tau)) \qquad a \in [\hat{a}..\hat{a} + n] \qquad a' \in [\hat{a}..\hat{a} + n]\end{array}}{A, M \to A, M}$$

B.3.3.2 Pointer relational comparison

Pointers $p = (\text{@}i, a)$ and $p' = (\text{@}i', a')$ may be compared with a relational operator (<, <=, etc.) if they have the same provenance ($i = i'$). The result is the Boolean result of the mathematical comparison of a and a'. To make this analogous to pointer subtraction a storage instance for i in A is alive, and both a and a' are within or one-past the footprint of that allocation. Otherwise UB. Note that this disallows relational comparison against null pointers.

For PNVI-ae-udi, this has to be adapted in much the same way as the pointer subtraction rule above.

$$\frac{\begin{array}{l}\text{[label: rel\_op\_ptrval}(p, p', \mathit{op}) = b\text{]} \\ p = (\text{@}i, a) \qquad p' = (\text{@}i', a') \qquad i = i' \qquad A(i) = (n, \_, \hat{a}, \_, \_, \_) \\ b = \mathit{op}(a, a') \qquad a \in [\hat{a}..\hat{a} + n] \qquad a' \in [\hat{a}..\hat{a} + n] \qquad \mathit{op} \in \{\leq, <, >, \geq\}\end{array}}{A, M \to A, M}$$

Relational comparison is used in practice between pointers to different objects. A variant which would allow that, called allow-inter-object-relational-operators true, removes the $i = i'$ test above and (in the zombie-pointers-become-indeterminate and zombie-pointers-allow-equality-only variants) additionally checks that i' maps to a live object with in-range address.

B.3.3.3 Pointer equality comparison

Pointers p and p' may unconditionally be compared with an equality operator (=, !=). The result is true if they are either both null or both non-null and have the same provenance and address; nondeterministically either $a = a'$ or `false` if they are both non-null and have different provenances; and false otherwise.

For PNVI-ae-udi, because equality comparison is permitted (without UB) irrespective of the provenances of the pointers, if the two pointers both have determined single provenances after looking up any symbolic IDs in A, this shall give `true`, otherwise the middle (nondeterministic) clause shall apply. The final A shall not resolve any symbolic IDs.

$$\frac{\begin{array}{l}\text{[label: eq\_op\_ptrval}(p, p') = b\text{]} \\ \begin{cases} b = \texttt{true} & \text{if } p = p' = \\ b = (a =_{\mathbb{Z}} a') & \text{if } \left(\begin{array}{l} (p = (\iota, a) \ \wedge \ p' = (\iota', a') \ \wedge \ A(\iota) = A(\iota') = \{i\}) \ \vee \\ (p = (\text{@}i, a) \ \wedge \ p' = (\text{@}j, a') \ \wedge i = j) \end{array} \right) \\ b \in \{(a =_{\mathbb{Z}} a'), \texttt{false}\} & \text{if } \left(\begin{array}{l} \left(\begin{array}{l} p = (\iota, a) \ \wedge \ p' = (\iota', a') \wedge \\ \neg(A(\iota) = A(\iota') = \{i\}) \end{array} \right) \vee \\ (p = (\pi, a) \ \wedge \ p' = (\pi', a') \ \wedge \ \pi \neq \pi') \end{array} \right) \\ b = \texttt{false} & \text{otherwise} \end{cases}\end{array}}{A, M \to A, M}$$

Note that the above nondeterminism appears to be necessary to admit the observable behavior of current compilers, but a simpler provenance-oblivious semantics is arguably desirable:

$$\frac{\begin{array}{l}\text{[label: eq\_op\_ptrval}(p, p') = b\text{]} \\ \begin{cases} b = \texttt{true} & \text{if } p = p' = \texttt{null} \\ b = \texttt{true} & \text{if } p = (\_, a), p' = (\_, a'), \text{and } a = a' \\ b = \texttt{false} & \text{otherwise} \end{cases}\end{array}}{A, M \to A, M}$$

These two options are called pointer-equality-provenance-nondet true and false.

B.3.3.4 Pointer array offset

Given a pointer p at C type τ, the result of offsetting p by integer x (either by array indexing or explicit pointer/integer addition) is as follows, where $x = n$ in PNVI*, or $x = (\pi', n)$ in PVI. For the operation to succeed, p is some non-null $(@i, a)$. Then there is a live storage instance for i, and the numeric result of the addition of $a + n * \text{sizeof}(\tau)$ is within or one-past the footprint of that storage instance. Otherwise the operation flags UB.

For PNVI-ae-udi, if p is ambiguous (i.e., $p = (\iota, a)$ and $A(\iota) = \{i_1, i_2\}$ then if x is non-zero this will only be defined behavior for (at most) one of the two, and then ι shall be resolved to that one in the final state. If $x = 0$ it does not resolve the ambiguity.

$$\text{iso\_array\_offset\_ptrval}(A, p, \tau, x) = \begin{cases} (@i, a') & \text{if } p = (@i, a) \text{ and} \\ & a' = a + n * \text{sizeof}(\tau) \text{ and} \\ & A(i) = (n'', \_, a'', \_, \_, \_) \text{ and} \\ & a' \in [a''..a'' + n''] \\ \text{UB: out of bounds} & \text{if all except the last conjunct} \\ & \text{above hold} \\ \text{UB: empty prov} & \text{if } p = (@\texttt{empty}, a) \\ \text{UB: killed prov} & \text{if } p = (@i, a) \text{ and } A(i) = \texttt{killed} \\ \text{UB: null pointer} & \text{if } p = \texttt{null} \end{cases}$$

B.3.3.5 Pointer member offset

Given a non-null pointer p at C type τ, which points to the start of a struct or union type object (ISO/IEC 9899:2018 suggests this has to exist, writing *"The value is that of the named member of the object to which the first expression points"*) with a member m, if p is (π, a), the result of offsetting the pointer to member m has the same provenance π and the suitably offset a.

If p is null, the result is a pointer with empty provenance and the integer offset of m within τ's representation (this is de facto C behavior, in the sense that the GCC torture tests rely on it; it does not exactly match ISO/IEC 9899:2018).

For the first case, in principle one might want p to point to the start of an object of type τ, with UB otherwise, but without a subobject-aware effective-type semantics, which is beyond the scope of this document, that cannot be checked. Instead, the definition just checks that there is a live storage instance of p's provenance such that the resulting address is within or one-past its a footprint. That makes this analogous to pointer array offset.

$$\text{member\_offset\_ptrval}(p, \tau, m) = \begin{cases} (\pi, a'), & \text{if } p = (@i, a) \text{ and} \\ & a' = a + \text{offsetof\_ival}(\tau, m) \text{ and} \\ & A(i) = (n'', \_, a'', \_, \_, \_) \text{ and} \\ & a' \in [a''..a'' + n''] \\ (@\texttt{empty}, \text{offsetof\_ival}(\tau, m)), & \text{if } p = \texttt{null}. \end{cases}$$

B.3.3.6 Casts (PNVI-plain)

In PNVI-plain, a cast of a pointer value p to an integer value (at type τ) just converts null pointers to zero and non-null pointer values to the address a of the pointer, if that is representable in τ, otherwise flagging UB. The provenance of the pointer is discarded. At present the definition requires that the object is live and that its address is within bounds.

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} 0, & \text{if } p = \texttt{null}; \\ a, & \text{if } p = (@i, a) \text{ and} \\ & A(i) = (n'', \_, a'', \_, \_, \_) \text{ and} \\ & a \in [a''..a'' + n''] \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

In PNVI-plain, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, it returns a pointer $(@i, x)$ with the provenance of that storage instance. (The "properly within" prevents the one-past ambiguous case.) If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \_) \text{ and } x \in [a..a+n-1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

B.3.3.7 Casts (PNVI-ae)

In PNVI-ae, the result of a cast of a pointer value p to an integer value is exactly as in PNVI-plain. In addition, for a cast of pointer value $p = (@i, a)$ with provenance $@i$, where $A(i) = (n, \tau_{\text{opt}}, a, f, k, t)$ is the storage instance metadata for i, the memory state (A, M) is updated to $(A(i \mapsto (n, \tau_{\text{opt}}, a, f, k, \texttt{exposed})), M)$ to mark the that storage instance as exposed.

In PNVI-ae, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, and storage instance i is exposed, it returns a pointer $(@i, x)$ with the provenance of that storage instance. If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a+n-1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

B.3.3.8 Casts (PNVI-ae-udi)

In PNVI-ae-udi, a cast of a pointer value p to an integer is just like PNVI-ae.

Unlike PNVI-ae, PNVI-ae-udi permits a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer check that x is properly within the footprint of the storage instance by a check that it is properly within or one-past:

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a+n] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

But then a PNVI-ae-udi cast of an integer value to a pointer might create a pointer with ambiguous provenance (as in the definition of repr) : if it could be within or one-past two live storage instances, with IDs i_1 and i_2, and both storage instances have been marked as exposed, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This may only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second.

B.3.3.9 Casts (PVI)

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = (@\texttt{empty}, 0) \\ (\pi, n), & \text{otherwise, where } x = (\pi, n) \end{cases}$$

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} (@\texttt{empty}, 0), & \text{if } p = \texttt{null}; \\ (\pi, a), & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

B.3.3.10 Integer operations (PVI)

In PVI one also has to define the provenance results of all the other operations returning integer values.

These are given below for the basic operations, though they would also be needed for all the integer-returning library functions. Most would give integers with empty provenance. One might or might not also want to require that the objects of those provenances are live.

$$
\begin{aligned}
\pi \oplus \pi' &= \begin{cases} \pi, & \text{if } \pi = \pi' \text{ or } \pi' = \texttt{@empty}; \\ \pi', & \text{if } \pi = \texttt{@empty}; \\ \texttt{@empty}, & \text{otherwise.} \end{cases} \\
\text{op\_ival}(op, (\pi, n), (\pi', m)) &= (\pi \oplus \pi', op(n, m)), \text{ where } op \in \{+, *, /, \%, \&, |, \wedge\} \\
\text{op\_ival}(-, (\pi, n), (\pi', m)) &= \begin{cases} (\texttt{@empty}, n - m), & \text{if } \pi = \texttt{@}i \text{ and } \pi' = \texttt{@}i', \text{whether } i = i' \text{ or not;} \\ (\texttt{@}i, n - m), & \text{if } \pi = \texttt{@}i \text{ and } \pi' = \texttt{@empty}; \\ (\texttt{@empty}, n - m), & \text{if } \pi = \texttt{@empty}. \end{cases} \\
\text{eq\_ival}((\pi, n), (\pi', m)) &= (n = m) \\
\text{lt\_ival}((\pi, n), (\pi', m)) &= (n < m) \\
\text{le\_ival}((\pi, n), (\pi', m)) &= (n \leq m)
\end{aligned}
$$

B.3.4 Provenance of other operations

In addition to the operations defined above, some operations are desugared/elaborated to simpler expressions by the Cerberus pipeline. Their PVI results have provenance as follows; their PNVI* results are the same except that there integers have no provenance:

- the result of address-of (`&`) has the provenance of the object associated with the lvalue, for non-function-pointers, or empty for function pointers.
- prefix increment and decrement operators follow the corresponding pointer or integer arithmetic rules.
- the conditional operator has the provenance of the second or third operand as appropriate; simple assignment has the provenance of the expression; compound assignment follows the pointer or integer arithmetic rules; the comma operator has the provenance of the second operand.
- integer unary `+`, unary `-`, and `~` operators preserve the original provenance; logical negation `!` has a value with empty provenance.
- **`sizeof`** and **`_Alignof`** operators give values with empty provenance.
- bitwise shifts has the provenance of their first operand.
- Atomic operations have to have their own specific provenance properties, not discussed here, as do some library functions.

Annex C
(normative)

Differences to ISO/IEC 9899:2018

Implementations that conform to this document, shall behave as if the differences from ISO/IEC 9899:2018 described in this annex were applied to ISO/IEC 9899. This annex is organized as follows:

- Each page corresponds to a page with such differences in ISO/IEC 9899:2018.
- Page numbers in the footer correspond to the page numbering within this document, here.
- Page numbers in the inner footer correspond to an approximation of the page number in ISO/IEC 9899:2018.
- Additions to the text are marked as shown.
- Deletions of text are marked as ~~shown~~.
- If possible, numbers of clauses refer to the clauses of ISO/IEC 9899:2018.
- Two new sub-clauses are introduced in clause 3 with numbers 3.17 (“pointer provenance”) and 3.20 (“storage instance”). The given context of ISO/IEC 9899:2018 and the numbering indicates the places of insertion.
- The subclause (here 3.21.2) “~~indeterminate value~~” is modified and renamed to “indeterminate representation”.
- The subclause (here 3.21.4) “~~trap representation~~” is modified and renamed to “non-value representation”.
- Clause 6.2.4 of ISO/IEC 9899:2018 is renamed from ”~~Storage durations of objects~~” to ”Storage durations and object lifetimes”.
- Clause 7.22.3 of ISO/IEC 9899:2018 is renamed from ”~~Memory management functions~~” to ”Storage management functions”.

contains four separate memory locations: The member `a`, and bit-fields `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 **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

3.17

1 **pointer provenance**

provenance

an entity that is associated to a pointer value in the abstract machine, which is either empty, or the identity of a storage instance

3.18

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

3.19

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 Annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

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

3.20

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 (7.22.3) 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.

differences to ISO/IEC 9899:2018, § 3.20 page 6 **General**

3.21

1 **value**

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

3.21.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.21.2

1 **indeterminate representation**

~~either~~ object representation that either represents an unspecified value or ~~a trap~~ is a non-value representation

3.21.3

1 **unspecified value**

valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance ~~An unspecified value cannot be a trap representation.~~

3.21.4

1 **non-value representation**

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

3.21.5

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.[2)]

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

3.22

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.

3.23

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.

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

of those operations are all *side effects*,[12] 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.[13] 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 accessing a volatile 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~~ representation 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:

— Accesses to volatile 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 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.

[12] 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.

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

— 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)[19)]

— 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 struct-declaration-list

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 Sizes of integer types <limits.h>

1 The values given below shall be replaced by constant expressions. If the value and promoted type is in the range of the type **intmax_t** (for a signed type) or **uintmax_t** (for an unsigned type), see 7.20.1.5, the expression shall be suitable for use in **#if** preprocessing directives.

Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

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

[19)] See "future language directions" (6.11.3).

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), 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,~~[33)] has a constant address,[34)] if any, and retains its last-stored value throughout its lifetime.[35)] ~~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 7.22.3.

3 ~~An~~ The storage instance of an object whose identifier is declared without the storage-class specifier **_Thread_local**, and either with external or internal linkage or with the storage-class specifier **static**, has *static storage duration* ~~. Its~~ , as do storage instances for string literals and some compound literals.[36)] The object's lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 ~~An~~ The storage instance of an object whose identifier is declared with the storage-class specifier **_Thread_local** has *thread storage duration*. ~~Its~~ The object's 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 is a distinct ~~object~~ instance of the object and distinct associated storage instance 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.

5 ~~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 storage instances of temporary objects and some compound literals. 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.

6 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~~ The initial representation 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~~ representation of the object becomes indeterminate each time the declaration is reached.

7 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.[37)] If the scope is entered recursively, a new instance of the object and associated storage is created each time. ~~The initial value~~ The initial representation of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a

[33)] String literals, compound literals or certain objects with temporary lifetime may share a storage instance with other such objects.

[34)] 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.

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

[36)] Such are for example compound literals that are evaluated in file scope or that are **const** qualified and have only constant expressions as initializers.

[37)] 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.

optionally specified name and possibly distinct type.

— A *function type* describes a function with specified return type. A function type is characterized by its return type and the number 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 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; its 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. A pointer-to-function is valid if it refers to a valid function defintion of the program. Pointers additionally may have a special value *null* that is different from the address of any storage instance and has no provenance (for object pointers),[49] or from the address of any function of the program (for function pointers). If a pointer value is neither valid nor null, it is *invalid* . 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.[50] 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 not be used in ways that require (in different usages) more than one of these provenances

— An *atomic type* describes the type designated by the construct **_Atomic**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

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*.[51]

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.

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,[52] corresponding to the combinations of one, two, or all three of the **const**, **volatile**,

[49] 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.

[50] The provenance of a pointer value and the property that such a pointer value is valid or not are generally not observable. In particular, in the course of the same program execution the same pointer object with the same representation bytes (6.2.6) may sometimes represent valid values but with different provenance (and thus refer to different objects). Sometimes the object representation may even be indeterminate, namely when the lifetime of the storage instance has ended and no new storage instance uses the same address.. Yet, this information is part of the abstract state machine and may restrict the set of operations that can be performed on the pointer.

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

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

and **restrict** 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.[53] 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, 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.

28 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[53] 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 whether other groups of pointer types have the same representation or alignment requirements.[54]

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).

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 An addressable storage instance[55] of size m provides access to a byte array of length m. Each byte of the array has an *abstract address* , which is a value of type **uintptr_t** that is determined in an implementation-defined manner by pointer-to-integer conversion. 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 immediately* if they are equal. If the lifetime of any two distinct addressable storage instances X and Y overlaps, 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.[56]

3 The object representation of a pointer object does not necessarily determine provenance of a pointer value; at different points of the program execution, identical object representations of pointer values may refer to distinct storage instances. Unless stated otherwise, a storage instance becomes *exposed* when a pointer value p of effective type T* with this provenance is used in the following

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

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

[55] All storage instances that do not originate from an object definition with **register** storage class 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).

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

contexts:[57)]

— Any byte of the object representation of `p` is used in an expression.[58)]

— The byte array pointed-to by the first argument of a call to the **`fwrite`** library function intersects with an object representation of `p`.

— `p` is converted to an integer.

— `p` is used as an argument to a `%p` conversion specifier of the **`printf`** family of library functions.[59)]

Nevertheless, 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 does not thereby become exposed.[60)] 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:[61)]

— 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.

— The object representation of `p` intersects with a byte array pointed-to by the first argument of a call 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 invalid.

5 Except for bit-fields, 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 of type **`unsigned char`** shall be represented using a pure binary notation.[62)]

7 Values stored in non-bit-field objects of any other object type ~~consist of~~ are represented using $n \times$ **`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 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~~

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

[58)] 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 reading the representation of a pointer value `p` through a **`union`** with a type that is not a pointer type (for example an integer type) or with a pointer type that has a different object representation than the original pointer.

[59)] Passing a pointer value to a `%s` conversion, does not expose the storage instance.

[60)] 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.

[61)] Synthesized pointer values may alias in ways that cannot be predicted by simple data flow analysis.

[62)] 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 $2^{\texttt{CHAR\_BIT}} - 1$.

~~**memcpy**); the resulting set of bytes~~ bytes determine the value of the object; the position of the first byte of these in the byte array is the *byte offset* of the object in its storage instance, the converted address is called the *byte address* of the object, and the range of bytes within the byte array is called the *object representation* of the value. The object representation may be used to copy the value of the object into another object (e.g., by **memcpy**). Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the ~~set~~ range of m bits the bit-field comprises in the addressable storage unit holding it. 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.[63] Such a representation is called a ~~trap representation.~~ non-value 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.[64] The ~~value~~ object representation of a structure or union object is never a ~~trap~~ non-value representation, even though the ~~value of~~ byte range corresponding to a member of the structure or union object may be a ~~trap representation~~ non-value representation for that member.

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.[65] 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~~ non-value representation shall not be generated.

12 Loads and stores of objects with atomic types are done with **memory_order_seq_cst** semantics.

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), integer types capable of holding object pointers (7.20.1.4), input/output (7.21).

6.2.6.2 Integer types

1 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). 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.[66]

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. There need not be any padding bits; **signed char** shall not have any padding bits. There shall be exactly one 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 there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

[63] 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.

[64] Thus, for example, structure assignment need not copy any padding bits.

[65] 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 `==` is defined for type `T`, then `x == y` does not imply that **memcmp**`(&x, &y, sizeof (T))== 0`. Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

[66] Some combinations of padding bits might generate ~~trap~~ non-value representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a ~~trap~~ non-value 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.

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a ~~trap~~ non-value representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

— the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that produce such a value;

— the `+`, `-`, `*`, `/`, and `%` operators where one operand is a negative zero and the result is zero;

— compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

4 If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with operands that would produce such a value is undefined.

5 The values of any padding bits are unspecified.[67] A valid ~~(non-trap)~~ object representation of a signed integer type that represents a value 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.

6 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The *width* of an integer type is the same but including any sign bit; thus for unsigned integer types the two values 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 6.7.6 for declarators.[68] 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 have the same widths. 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:

[67] 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~~ non-value representation other than as part of an exceptional condition such as an overflow. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

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

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.

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.[77)]

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression (with an object type other than **void**) that potentially designates an object;[78)] 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 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.

2 Except when it is the operand of the **sizeof** 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 behavior is undefined if the lvalue has an incomplete type~~and does not have array type, the behavior is undefined. If~~, if the object representation is a non-value representation for the type,[79)] or if 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 with an initializer and no assignment to it has been performed prior to use)~~, the~~.

3 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 a byte address within (or one-past) the object representation of an exposed storage instance, such that the exposure happened before this lvalue conversion, and the result has that address and provenance.[80)]

The behavior is undefined if the pointer object has an indeterminate representation, in particular if the lvalue conversion does not happen during the lifetime of the provenance that was associated to the stored pointer value, the represented address is not a valid address (or one-past) for the associated provenance, or the represented address is not correctly aligned for the type.

4 Except when it is the operand of the **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

[77)] The cast and assignment operators are still required to remove extra range and precision.

[78)] 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.

[79)] Character types have no non-value representation, thus reading representation bytes of an addressable live storage instance is always defined.

[80)] If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

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.

5 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator,[81] 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*".

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions <stddef.h> (7.19), initialization (6.7.9), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and **_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.)

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 a *null pointer constant*.[82] 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.

4 Conversion of a null pointer to another 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 value could have been the result of the conversion of a null pointer value. The result is a null pointer.
- The operand value 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.[83]
- The pointer value is invalid.

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~~ might be invalid, and might produce an indeterminate representation when stored into an object.~~The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.~~

6 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 type~~ values.[84] If the pointer value is valid, its provenance is henceforth exposed. Except as previously specified, the result is the abstract address (which has

[81] 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.

[82] The macro **NULL** is defined in <stddef.h> (and other headers) as a null pointer constant; see 7.19.

[83] If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

[84] It is recommended that 0 is a member of that set.

type **uintptr_t**) converted to the target type. If the target type has a width that is less than the width of **uintptr_t** , the behavior is undefined. ~~The result need not be in the range of values of any integer type~~ If the target type is a signed type and the abstract address is larger than the maximum value of that type the implementation-defined conversion from **uintptr_t** to the target type as specified in 6.3.1.3 is applied.[85] If the pointer is null or valid, the integer result converted back to the pointer type shall compare equal to the original pointer.[86] For two valid pointer values that compare equal, conversion to the same integer type yields identical values.

7 A pointer to an object type may be converted to a pointer to a different object type, retaining its provenance. If the resulting pointer is not correctly aligned[87] 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.

8 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: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.16.1).

9 **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 are 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 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`).

[85] Thus, the result is an implementation-defined value or an implementation-defined signal is raised.

[86] 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.

[87] 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.

values.[98] 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"
```

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 <stddef.h> (7.19), the **mbstowcs** function (7.22.8.1), 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[99]

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

[98] This allows implementations to share storage instances for string literals and constant compound literals (6.5.2.5) with the same or overlapping representations.

[99] These tokens are sometimes called “digraphs”.

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value,[104] 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.[105]

3 The grouping of operators and operands is indicated by the syntax.[106] Except as specified later, side effects and value computations of subexpressions are unsequenced.[107]

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 is the declared type of the object, if any.[108] If a value is stored into an object having no declared type 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 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. For all other accesses to an object having no declared type, 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 following types:[109]

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

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

[105] This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

[106] 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 `?:` (6.5.15).

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.

[107] 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.

[108] ~~Allocated objects have~~ An object with allocated storage duration has no declaration and thus no declared type.

[109] The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;

— both types are pointers to qualified or unqualified versions of a character type or **`void`**.

7 If the expression that denotes the called 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.

8 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.

9 If the 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, the behavior is undefined.

10 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 is indeterminately sequenced with respect to the execution of the called function.[116)]

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

12 **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 (including prototypes) (6.7.6.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1 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.

2 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

3 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,[117)] 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.

4 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

[116)] In other words, function executions do not “interleave” with each other.

[117)] 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~~ non-value representation.

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.[118] 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.

5 Accessing a member of an atomic structure or union object results in undefined behavior.[119]

6 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.

7 **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.

8 **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
```

9 **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)
            /* ... */
```

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)
```

[118] 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`.

[119] 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.

list.[121)]

4 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, 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.

5 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.

6 All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.[122)]

7 String literals, and compound literals with const-qualified types, need not designate distinct objects.~~This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.~~ [123)]

8 **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.

9 **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.

10 **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});
```

11 **EXAMPLE 4** A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

12 **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.

[121)] 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.

[122)] For example, subobjects without explicit initializers are initialized to zero.

[123)] This allows implementations to share storage instances for string literals and constant compound literals with the same or overlapping representations.

13 **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"
```

might yield 1 if the literals' storage instance is shared.

14 **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);
```

15 **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;
}
```

The function `f()` always returns the value 1.

16 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~~representation. The behavior of the lvalue conversion of `p` in the assignment to `q` would then be undefined.

Forward references: type names (6.7.7), initialization (6.7.9).

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 (*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.

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

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.

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.

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*".[124)]

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

[124)] 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, ~~and~~ the address of an object after the end of its lifetime, or any other invalid value.

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 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.[128)]

9 When two pointers are subtracted, both shall be valid and point to elements of the same array object, or one past the last element of the array object;[129)] 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 `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. ~~In other words, if the~~

[128)] 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).

[129)] This implies that they also have the same provenance.

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.~~Another way to approach pointer arithmetic is first to convert the pointer(s) to character 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 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.~~

12 **NOTE 3** Another way to approach pointer arithmetic is first to convert the pointer(s) to character 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 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.

```
{
      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
}
```

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 (6.7.6.2), common definitions `<stddef.h>` (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` 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` 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

`E1`/2^{E2}. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

6.5.8 Relational operators

Syntax

1 *relational-expression:*
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-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 if the specified relation is true and 0 if it is false.[130)] The result has type **`int`**.

6.5.9 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;

— both operands are pointers to qualified or unqualified versions of compatible types;

[130)] 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`".

— 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.[131] None of the operands shall be an invalid pointer value. Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type **int**. For any pair of operands, exactly one of the relations is 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 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.

6 ~~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. Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.~~ objects and compare equal if and only if they have the same abstract address.

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.

6.5.10 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).

[131] Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.


```
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 *
```

6.5.16 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,[133)] but is not an lvalue. The type of an assignment 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.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[134)]

— 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;

[133)] 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.

[134)] 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**).

6.7 Declarations

Syntax

1 *declaration:*
declaration-specifiers *init-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration
declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{\text{opt}}$
type-specifier *declaration-specifiers*$_{\text{opt}}$
type-qualifier *declaration-specifiers*$_{\text{opt}}$
function-specifier *declaration-specifiers*$_{\text{opt}}$
alignment-specifier *declaration-specifiers*$_{\text{opt}}$
init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*
init-declarator:
declarator
declarator **=** *initializer*

Constraints

2 A declaration other than a static_assert 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 attributes 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;[140)]

— 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 that indicate the linkage, storage duration, and part of the type of the entities that 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.

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 (including in prototypes), it is the adjusted type (see 6.7.6.3) that is required to be complete.

Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.9), type names (6.7.7), type qualifiers (6.7.3).

[140)] Function definitions have a different syntax, described in 6.9.1.

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

typedef
extern
static
_Thread_local
auto
register

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that **_Thread_local** may appear with **static** or **extern**.[141)]

3 In the declaration of an object with block scope, if the declaration specifiers include **_Thread_local**, they shall also include either **static** or **extern**. If **_Thread_local** appears in any declaration of an object, it shall be present in every declaration of that object.

4 **_Thread_local** shall not appear in the declaration specifiers of a function declaration.

Semantics

5 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4 .

6 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.[142)]

7 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

8 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.

Forward references: type definitions (6.7.8).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier

[141)] See "future language directions" (6.11.5).

[142)] The implementation can treat any **register** declaration simply as an **auto** declaration. However, whether or not ~~addressable~~ a storage instance that would otherwise be addressable is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary & operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

differences to ISO/IEC 9899:2018, § 6.7.2 page 84 **Language**

incomplete until immediately after the } that terminates the list, and complete thereafter.

9 A member of a structure or union may have any complete object type other than a variably modified type.[144] 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 *bit-field*;[145] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[146] 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**.

11 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 or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[147] 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.

13 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. This applies recursively if the containing structure or union is also anonymous.

14 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

15 Within a structure object, the non-bit-field members and the units in which bit-fields reside 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), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

16 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), and vice versa.

17 There may be unnamed padding at the end of a structure or union.

18 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 ->) 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.

[144] A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[145] 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.

[146] As specified in 6.7.2 above, if the actual type specifier used is **int** or a typedef-name defined as **int**, then it is implementation-defined whether the bit-field is signed or unsigned.

[147] An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.


```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

24 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 = 42;         // valid
dp = &(s2->d[0]); // valid
*dp = 42;         // undefined behavior
```

25 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~~are set to an indeterminate representation, that may or may not coincide with a copy of the representation of the elements of the source array.

26 **EXAMPLE 3** 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 (6.7.6), tags (6.7.2.3).

6.7.2.2 Enumeration specifiers

Syntax

1 *enum-specifier:*
enum *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**
enum *identifier*$_{\text{opt}}$ **{** *enumerator-list* **, }**
enum *identifier*
enumerator-list:
enumerator
enumerator-list **,** *enumerator*
enumerator:
enumeration-constant
enumeration-constant **=** *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**.

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 (6.7.6), type definitions (6.7.8).

6.7.2.4 Atomic type specifiers

Syntax

1 *atomic-type-specifier:*

_Atomic (*type-name* **)**

Constraints

2 Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.8.3).

3 The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, or a qualified type.

Semantics

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.

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

const
restrict
volatile
_Atomic

Constraints

2 Types other than pointer types whose referenced type is an object type shall not be restrict-qualified.

3 The **_Atomic** qualifier shall not be used if the implementation does not support atomic types (see 6.10.8.3).

4 The type modified by the **_Atomic** qualifier shall not be an array type or a function type.

Semantics

5 The properties associated with qualified types are meaningful only for expressions that are lvalues.[153)]

6 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.

7 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

[153)] The implementation can place a **const** object that is not **volatile** in a read-only ~~region of~~ storage instance. Moreover, ~~the implementation need not allocate~~ a storage instance for such an object need not be addressable if its address is never used.

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.

6 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.

7 **EXAMPLE 1**

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

8 **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.

9 **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
      r = c;      // compatible, but defined behavior only if
                  // n == 6 and m == n+1
}
```

10 **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**, **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). 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
      };
      int D[m];                   // valid:  auto VLA
      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
```


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 initializer specifies the initial value stored in an object.

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~~ representation even after initialization.

10 If an object that has automatic storage duration is not initialized explicitly, its ~~value~~ representation 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. 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.

differences to ISO/IEC 9899:2018, § 6.7.9 page 106 **Language**

6.8 Statements and blocks

Syntax

1 *statement:*

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

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~~ the representation of objects without an initializer becomes indeterminate) 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:*

identifier **`:`** *statement*
`case` *constant-expression* **`:`** *statement*
`default` **`:`** *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement. Further constraints on such labels are discussed under the **`switch`** statement.

3 Label names shall be unique within a function.

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~~object with an indeterminate representation. Similarly, the call to the function `f` cannot be reached.

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*

`while (` *expression* **`)`** *statement*
`do` *statement* **`while (`** *expression* **`) ;`**
`for (` $expression_{opt}$ **`;`** $expression_{opt}$ **`;`** $expression_{opt}$ **`)`** *statement*
`for (` *declaration* $expression_{opt}$ **`;`** $expression_{opt}$ **`)`** *statement*

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`**.

Semantics

4 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[176)]

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,[177)] and none of the following operations are performed in its body, controlling expression or (in the case of a **`for`** statement) its *expression-3*:[178)]

— input/output operations

— accessing a volatile object

— synchronization or atomic operations.

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.

[176)] 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 *clause-1* of a **`for`** statement.

[177)] An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

[178)] This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.

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.

3 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), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 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.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. 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), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[182)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
declaration-specifiers declarator declaration-list$_{\text{opt}}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

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.[183)]

3 The return type of a function shall be **void** or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier. No declaration list shall follow.

[182)] Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

differences to ISO/IEC 9899:2018, § 6.9.1 page 118 **Language**

6 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations.

Semantics

7 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[184] the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

8 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.

9 Each parameter has automatic storage duration; its identifier is an lvalue.~~A parameter identifier cannot be redeclared in the function body except in an enclosed block.~~ ~~The layout of the storage for parameters is unspecified.~~[185]

10 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.)

11 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

12 Unless otherwise specified, if the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

13 **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; }
```

[183] The intent is that the type category in a function definition 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
int (*fp)(void);                // fp points to a function that has type F
F *Fp;                          // Fp points to a function that has type F
```

[184] See "future language directions" (6.11.7).

[185] 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.

— 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.[208] 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.[209]

— 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.[210]

— 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.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.[211]

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.[212] Implementations may share their own

[208] 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.

[209] Such macros might not contain the sequence points that the corresponding function calls do.

[210] 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.

[211] Thus, a signal handler cannot, in general, call standard library functions.

[212] 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.

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.[213)]

7 Unless otherwise specified, library functions by themselves do not expose storage instances, but library functions that execute application specific callbacks[214)] 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);
```

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.

[213)] This allows implementations to parallelize operations if there are no visible side effects.

[214)] 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).

7.5 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[225] 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 indeterminatevalue~~initially representation of the object corresponding to **errno** in any other thread is indeterminate), but is never set to zero by any library function.[226] 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,[227] may also be specified by the implementation.

[225] 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**()).

[226] 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.

[227] See "future library directions" (7.31.3).

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[272)] 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[273)] have state, as of the time the **longjmp** function was called, except that the ~~values~~ representation 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~~ is 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
}
```

[272)] 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.

[273)] This includes, but is not limited to, the floating-point status flags and the state of open files.

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 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 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~~ object designated by **`errno`** ~~is indeterminate~~ has an indeterminate representation.[276)]

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`**.

Forward references: the **`abort`** function (7.22.4.1), the **`exit`** function (7.22.4.4), the **`_Exit`** function (7.22.4.5), the **`quick_exit`** function (7.22.4.7).

7.14.2 Send signal

7.14.2.1 The `raise` function

Synopsis

1
```
#include <signal.h>
int raise(int sig);
```

[276)] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

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 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~~ representation of ap in the calling function is indeterminate and shall be passed to the **va_end** macro prior to any further reference to ap.[277)]

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.

[277)] 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.

7.17.2 Initialization

7.17.2.1 The `ATOMIC_VAR_INIT` macro

Synopsis

1
```
#include <stdatomic.h>
#define ATOMIC_VAR_INIT(C value)
```

Description

2 The **`ATOMIC_VAR_INIT`** macro 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~~has initially an indeterminate representation; however, the default (zero) initialization for objects with static or thread-local storage duration is guaranteed to produce a valid state.[279)]

3 Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

4 **EXAMPLE**

```
atomic_int guide = ATOMIC_VAR_INIT(42);
```

7.17.2.2 The `atomic_init` generic function

Synopsis

1
```
#include <stdatomic.h>
void atomic_init(volatile A *obj, C value);
```

Description

2 The **`atomic_init`** generic function 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`** generic function returns no value.

6 **EXAMPLE**

```
atomic_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:[280)]

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

[279)] See "future library directions" (7.31.8).
[280)] See "future library directions" (7.31.8).

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));
```

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 operations are applicable to an object of any atomic integer type. None of these operations is applicable to **`atomic_bool`**. The key, operator, and computation correspondence is:

key	op	computation
add	+	addition
sub	-	subtraction
or	|	bitwise inclusive or
xor	^	bitwise exclusive or
and	&	bitwise and

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);
```

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 two's complement representation with 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.

5 **NOTE** 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 are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated value of the object, whereas the value returned by the `atomic_fetch` and modify generic functions is the previous value of the atomic object.

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.

2 Operations on an object of type **`atomic_flag`** shall be lock free.

3 **NOTE** 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.

4 The macro **`ATOMIC_FLAG_INIT`** may be used to initialize an **`atomic_flag`** to the clear state. An **`atomic_flag`** that is not explicitly initialized with **`ATOMIC_FLAG_INIT`** ~~is initially in an indeterminate state~~has initially an indeterminate representation.

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.[285] 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,[286] <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").

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 Exact-width integer types

1 The typedef name **int***N***_t** designates a signed integer type with width *N*, no padding bits, and a two's complement representation. 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 standard or extended integer types with ~~widths of 8, 16, 32, or 64 bits~~a particular width, no padding bits, and (for the signed types) that have a two's complement representation, it shall define the corresponding typedef names.

7.20.1.2 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 The following types are required:

[285] See "future library directions" (7.31.10).

[286] Some of these types might denote implementation-defined extended integer types.

differences to ISO/IEC 9899:2018, § 7.20.1.2 page 218 **Library**

int_least8_t	**uint_least8_t**
int_least16_t	**uint_least16_t**
int_least32_t	**uint_least32_t**
int_least64_t	**uint_least64_t**

All other types of this form are optional.

7.20.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[287] 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~~ the corresponding 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.~~ required.

2 **NOTE 1** The types **intptr_t** and **uintptr_t** are possibly wider than the types **intmax_t** and **uintmax_t** (7.20.1.5). This exception is intended to accommodate implementations that otherwise would not be able to specify **intptr_t** and **uintptr_t** consistent with the rules for these types.

3 **NOTE 2** Although these integer types allow roundtrip conversions of values of type pointer to **void** and therefore guarantee that such conversions do not lose information, arithmetic on these types is not necessarily consistent with arithmetic on pointer to character types, nor can properties of pointer values such as alignment be portably deduced from the bit pattern of the integer result of a conversion.

4 On the other hand, the rules for abstract addresses in 6.2.6.1, 6.5.8 and 6.5.9 impose that two values of type **uintptr_t** that originate from conversions of two pointers to the same storage instance compare the same for the relational and equality operators as the original pointer values. Also, the reconstruction of all the bits of a valid abstract address that has previously been exposed gives rise to an integer value that converts back to the corresponding byte address.

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 with the possible exception of signed extended integer types that are wider than **long long** and that are referred by the type definition for an exact width integer type or for **intptr_t** :

intmax_t

[287] 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.

The following type designates ~~an unsigned integer type capable of representing any value of any~~ the unsigned integer type ~~:~~ that corresponds to **intmax_t**:[288)]

uintmax_t

These types are required.

7.20.2 Limits of specified-width integer types

1 The following object-like macros specify the minimum and maximum limits 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, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign, except where stated to be exactly the given value.

7.20.2.1 Limits of exact-width integer types

1 — minimum values of exact-width signed integer types

INT*N*_MIN	exactly $-(2^{N-1})$

— maximum values of exact-width signed integer types

INT*N*_MAX	exactly $2^{N-1} - 1$

— maximum values of exact-width unsigned integer types

UINT*N*_MAX	exactly $2^{N} - 1$

7.20.2.2 Limits of minimum-width integer types

1 — minimum values of minimum-width signed integer types

INT_LEAST*N*_MIN	$-(2^{N-1} - 1)$

— maximum values of minimum-width signed integer types

INT_LEAST*N*_MAX	$2^{N-1} - 1$

— maximum values of minimum-width unsigned integer types

UINT_LEAST*N*_MAX	$2^{N} - 1$

7.20.2.3 Limits of fastest minimum-width integer types

1 — minimum values of fastest minimum-width signed integer types

INT_FAST*N*_MIN	$-(2^{N-1} - 1)$

— maximum values of fastest minimum-width signed integer types

INT_FAST*N*_MAX	$2^{N-1} - 1$

[288)] Thus this type is capable of representing any value of any unsigned integer type with the possible exception of particular extended integer types that are wider than unsigned **long long**.

differences to ISO/IEC 9899:2018, § 7.20.2.3 page 220 **Library**

SIZE_MAX	65535

— limits of **wchar_t**

WCHAR_MIN	*see below*
WCHAR_MAX	*see below*

— limits of **wint_t**

WINT_MIN	*see below*
WINT_MAX	*see below*

3 If **sig_atomic_t** (see 7.14) is defined as a signed integer type, the value of **SIG_ATOMIC_MIN** shall be no greater than -127 and the value of **SIG_ATOMIC_MAX** shall be no less than 127; otherwise, **sig_atomic_t** is defined as an unsigned integer type, and the value of **SIG_ATOMIC_MIN** shall be 0 and the value of **SIG_ATOMIC_MAX** shall be no less than 255.

4 If **wchar_t** (see 7.19) is defined as a signed integer type, the value of **WCHAR_MIN** shall be no greater than -127 and the value of **WCHAR_MAX** shall be no less than 127; otherwise, **wchar_t** is defined as an unsigned integer type, and the value of **WCHAR_MIN** shall be 0 and the value of **WCHAR_MAX** shall be no less than 255.[290)]

5 If **wint_t** (see 7.29) is defined as a signed integer type, the value of **WINT_MIN** shall be no greater than -32767 and the value of **WINT_MAX** shall be no less than 32767; otherwise, **wint_t** is defined as an unsigned integer type, and the value of **WINT_MIN** shall be 0 and the value of **WINT_MAX** shall be no less than 65535.

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 7.20.1.2 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. If the value and promoted type is in the range of the type **intmax_t** (for a signed type) or **uintmax_t** (for an unsigned type), see 7.20.1.5, the expression is suitable for use in **#if** preprocessing directives

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**:

[290)] The values **WCHAR_MIN** and **WCHAR_MAX** do not necessarily correspond to members of the extended character set.

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 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 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.)[293)]

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~~may henceforth not consist of valid multibyte characters.

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 number

[293)] The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

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 are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a 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 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~~ lifetime of a **FILE** object ~~is indeterminate after~~ ends when 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 of a **FILE** object need not serve in place of the original.

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.[294)]

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

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

[294)] 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.

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);
```

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);
```

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;

_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[299] 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~~ members of the array at any time ~~are indeterminate~~have unspecified values.

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.[300]

[299] 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.

[300] The **fprintf** functions perform writes to memory for the `%n` specifier.

of 2, then the precision is sufficient to distinguish[305)] 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.[306)] Characters from the array arewritten 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 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.[307)]

`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.[308)] 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.

[305)] 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.

[306)] No special provisions are made for multibyte characters.

[307)] Redundant shift sequences can result if multibyte characters have a state-dependent encoding.

[308)] See "future library directions" (7.31.11).

`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).[312)]

If no `l` length modifier is present, thecorresponding 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. Thecorresponding 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.[312)]

If no `l` length modifier is present, thecorresponding 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. Thecorresponding 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 characters from a set of expected characters (the *scanset*).[312)]

If no `l` length modifier is present, thecorresponding 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. Thecorresponding 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 ~~an~~ the same implementation-defined set of sequences ~~, which should be the same as the 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`**. ~~The input item is converted to a pointer value in an implementation-defined manner.~~

[312)] 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.

– If the input ~~item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the conversion is undefined.~~sequence could have been printed from a null pointer value, a null pointer value is stored in `*ptr`.

– 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 representation of a valid pointer with address x and the provenance of that storage instance is synthesized in `*ptr`.[313)]

– Otherwise the representation of `*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.[314)]

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 (including new-line characters) is 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 **`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:

[313)] 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.

[314)] See “future library directions” (7.31.11).

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);
```

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.[315)]

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, ...)
{
      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);
```

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.[315)]

[315)] As the functions **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, and **vsscanf** invoke the **va_arg** macro, ~~the value of~~ arg after the return ~~is~~ has an indeterminate representation.

differences to ISO/IEC 9899:2018, § 7.21.6.9 page 248 **Library**

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**.[316)]

7.21.7.2 The fgets function

Synopsis

1
```
#include <stdio.h>
char *fgets(char * restrict s, int n,
      FILE * restrict 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.

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~~ members of the array have unspecified values 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 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 is appended to the output stream.

Returns

3 The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

7.21.7.4 The fputs function

Synopsis

1
```
#include <stdio.h>
int fputs(const char * restrict s,
      FILE * restrict stream);
```

Description

2 The **fputs** function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null character is not written.

Returns

3 The **fputs** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

[316)] An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

7.21.7.9 The `puts` function

Synopsis

1
```
#include <stdio.h>
int puts(const char *s);
```

Description

2 The **`puts`** function writes the string pointed to by `s` to the stream pointed to by **`stdout`**, and appends a new-line character to the output. The terminating null character is not written.

Returns

3 The **`puts`** function returns **`EOF`** if a write error occurs; otherwise it returns a nonnegative value.

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 specified by `c` (converted to an **`unsigned char`**) back onto the input stream pointed to by `stream`. Pushed-back 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 characters for the stream. The external storage corresponding to the stream is unchanged.

3 One character 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 were pushed back.[317)] 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 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~~ has an indeterminate representation after the call.[318)]

Returns

6 The **`ungetc`** function returns the character 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>
size_t fread(void * restrict ptr,
      size_t size, size_t nmemb,
      FILE * restrict stream);
```

[317)] Note that a file positioning function could further modify the file position indicator after discarding any pushed-back characters.

[318)] See "future library directions" (7.31.11).

Description

2 The **fread** function reads, into the array pointed to by ptr, up to nmemb elements 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 stored, in the order read, in 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 successfully read. If an error occurs, the resulting ~~value~~ representation of the file position indicator for the stream is indeterminate. If a partial element is read, its ~~value~~ representation 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);
```

Description

2 The **fwrite** function writes, from the array pointed to by ptr, up to nmemb elements 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 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 successfully written. If an error occurs, the resulting ~~value~~ representation 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);
```

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).


```
static unsigned long int next = 1;

int rand(void)   //  RAND_MAX assumed to be 32767
{
      next = next * 1103515245 + 12345;
      return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
      next = seed;
}
```

7.22.3 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~~ is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and ~~then~~ 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 ~~objects~~storage instance. The pointer returned points to the start ~~(lowest byte address)~~ address of the allocated ~~spaces~~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.[323)]

7.22.3.1 The aligned_alloc function

Synopsis

1
```
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

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~~ representation is indeterminate. 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.

7.22.3.2 The calloc function

[323)] 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 *calloc(size_t nmemb, size_t size);
```

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.[324)]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.3.3 The `free` function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
```

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.[325)] 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.

7.22.3.4 The `malloc` function

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

2 The **malloc** function allocates ~~space for an object~~ a storage instance whose size is specified by `size` and whose ~~value~~ representation is indeterminate.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.3.5 The `realloc` function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

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

[324)] Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

[325)] 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.

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 objects in the new storage instance that have hidden state and need explicit initialization (such as variable argument lists, atomic objects, mutexes, or condition variables) may have an indeterminate representation.
- Resources reserved for the original 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.

7.22.4 Communication with the environment

7.22.4.1 The abort function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void abort(void);
```

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.

7.22.4.2 The atexit function

Synopsis

1
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The **atexit** function registers the function pointed to by `func`, to be called without arguments at normal program termination.[326)] It is unspecified whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed.

[326)] 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.

7.22.7 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 function 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.[334)] Changing the **LC_CTYPE** category causes the internal object describing the conversion state of these functions to ~~be indeterminate~~ have an indeterminate representation.

7.22.7.1 The mblen function

Synopsis

1
```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

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);
```

3 The implementation shall behave as if no library function calls the **mblen** function.

Returns

4 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 (7.22.7.2).

7.22.7.2 The mbtowc function

Synopsis

1
```
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc,
      const char * restrict s,
      size_t n);
```

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

[334)] 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.

7.24 String handling <string.h>

7.24.1 String function conventions

1 The header <string.h> declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.[336] 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.

2 Where an argument declared as **size_t** n specifies 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 character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

3 For all functions in this subclause, each character 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 representation of a pointer object 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.

7.24.2.1 The memcpy function

Synopsis

1
```
#include <string.h>
void *memcpy(void * restrict s1,
      const void * restrict s2,
      size_t n);
```

Description

2 The **memcpy** function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **memcpy** function returns the value of s1.

7.24.2.2 The memmove function

Synopsis

1
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

2 The **memmove** function copies n characters from the object pointed to by s2 into the object pointed to by s1. Copying takes place as if the n characters from the object pointed to by s2 are first copied into a temporary array of n characters that does not overlap the objects pointed to by s1 and s2, and then the n characters from the temporary array are copied into the object pointed to by s1.

Returns

3 The **memmove** function returns the value of s1.

7.24.2.3 The strcpy function

Synopsis

1

[336] See "future library directions" (7.31.13).

Description

2 The **strncat** function 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 pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result.[338] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strncat** function returns the value of s1.

Forward references: the **strlen** function (7.24.6.3).

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 (both interpreted as **unsigned char**) that differ in the objects being compared.

7.24.4.1 The memcmp function

Synopsis

1
```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

2 The **memcmp** function compares the first n characters of the object pointed to by s1 to the first n characters of the object pointed to by s2.[339]

Returns

3 The **memcmp** 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.

7.24.4.2 The strcmp function

Synopsis

1
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description

2 The **strcmp** function compares the string pointed to by s1 to the string pointed to by s2.

Returns

3 The **strcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

7.24.4.3 The strcoll function

Synopsis

1
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

2 The **strcoll** function compares the string pointed to by s1 to the string pointed to by s2, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

[338] Thus, the maximum number of characters that can end up in the array pointed to by s1 is **strlen**(s1)+n+1.

[339] The ~~contents~~ values of ~~"holes"~~ bytes that are used as padding for purposes of alignment within structure objects ~~are indeterminate~~take on unspecified values when a value is stored in the object (cf. 6.2.6.1). Strings shorter than their allocated space and unions can also cause problems in comparison.

Returns

3 The **strcoll** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 when both are interpreted as appropriate to the current locale.

7.24.4.4 The strncmp function

Synopsis

1
```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

2 The **strncmp** function compares not more than n characters (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** function 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 function

Synopsis

1
```
#include <string.h>
size_t strxfrm(char * restrict s1,
      const char * restrict s2,
      size_t n);
```

Description

2 The **strxfrm** function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the **strcmp** function is applied to two transformed 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. No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. 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

3 The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, the contents of the members of the array pointed to by s1 ~~are indeterminate~~ have an indeterminate representation.

4 **EXAMPLE** The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by s.

```
1 + strxfrm(NULL, s, 0)
```

7.24.5 Search functions

7.24.5.1 The memchr function

Synopsis

1
```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Description

2 The **memchr** function locates the first occurrence of c (converted to an **unsigned char**) in the initial n characters (each interpreted as **unsigned char**) of the object pointed to by s. The implementation

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~~representation.

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.

`%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~~ members of the array ~~are indeterminate~~ have an indeterminate representation.

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[363] 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, andwritten 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.[364] 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.

[363] 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.

[364] See "future library directions" (7.31.16).

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. Thecorresponding 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, thecorresponding 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 ~~an~~ the same implementation-defined set of sequences ~~, which should be the same as the 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`**. ~~The input item is converted to a pointer value in an implementation-defined manner.~~

- ~~If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the conversion is undefined.~~sequence could have been printed from a null pointer value, a null pointer value is stored in `*ptr`.
- 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 representation of a valid pointer with address x and the provenance of that storage instance is synthesized in `*ptr`.[368)]
- Otherwise the representation of `*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.[369)]

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 (including new-line wide characters) is 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.

[368)] 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.

[369)] See "future library directions" (7.31.16).

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);
```

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.[370)]

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, ...)
{
      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);
      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);
```

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.[370)]

[370)] As the functions **vfwprintf**, **vswprintf**, **vfwscanf**, **vwprintf**, **vwscanf**, and **vswscanf** invoke the **va_arg** macro, the ~~value~~ representation of `arg` after the return is indeterminate.


```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

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 value of the macro **EILSEQ** is stored in **errno** and the **fgetwc** function returns **WEOF**.[371)]

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);
```

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~~ members have an indeterminate representation 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);
```

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**.

[371)] 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.

Description

2 The **wcscmp** function compares the wide string pointed to by s1 to the wide string pointed to by s2.

Returns

3 The **wcscmp** function 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.

7.29.4.4.2 The wcscoll function

Synopsis

1
```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **wcscoll** function compares the wide string pointed to by s1 to the wide string pointed to by s2, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Returns

3 The **wcscoll** function 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.

7.29.4.4.3 The wcsncmp function

Synopsis

1
```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
      size_t n);
```

Description

2 The **wcsncmp** function 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

3 The **wcsncmp** function 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.

7.29.4.4.4 The wcsxfrm function

Synopsis

1
```
#include <wchar.h>
size_t wcsxfrm(wchar_t * restrict s1,
      const wchar_t * restrict s2,
      size_t n);
```

Description

2 The **wcsxfrm** function 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 **wcscmp** 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 **wcscoll** 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

3 The **wcsxfrm** function 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~~ array

elements pointed to by `s1` ~~are indeterminate~~ have an indeterminate representation.

4 **EXAMPLE** 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`:

```
1 + wcsxfrm(NULL, s, 0)
```

7.29.4.4.5 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`.

7.29.4.5 Wide string search functions

7.29.4.5.1 The `wcschr` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

2 The **`wcschr`** function 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

3 The **`wcschr`** function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

7.29.4.5.2 The `wcscspn` function

Synopsis

1
```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **`wcscspn`** function 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

3 The **`wcscspn`** function returns the length of the segment.

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~~ members of the array ~~are indeterminate~~ have an indeterminate representation.

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.[379)]

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.

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`**.

[379)] 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.

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 certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (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 `()`, `&&`, `||`, `?:`, 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 (6.5.16).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

— 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 (6.7.6.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.9).

— ~~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 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 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 convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.22.1.3, 7.29.4.1.1).

— ~~The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, **realloc**, and **aligned_alloc** functions (??). 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 (7.22.3).

— Whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed (7.22.4.2).

— Whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed (7.22.4.3).

— Which of two elements that compare as equal is matched by the **bsearch** function (7.22.5.1).

— The order of two elements that compare as equal in an array sorted by the **qsort** function (7.22.5.2).

— 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 if the rounded value is outside the range of the return type (F.10.6.5, F.10.6.7).

— The sign of one part of the **complex** result of several math functions for certain special cases in IEC 60559 compatible implementations (G.6.1.1, G.6.2.2, G.6.2.3, G.6.2.4, G.6.2.5, G.6.2.6, G.6.3.1, G.6.4.2).

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 (5.2.1.2).

— 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 (~~**??**the object has an indeterminate representation (6.2.4, 6.7.9, 6.8).

— A ~~trap~~ non-value representation is read by an lvalue expression that does not have character type (6.2.6.1).

— A ~~trap~~ non-value 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).

— The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

— 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 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).

— 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 (7.22.3).

— The value of a pointer that refers to ~~space~~ a storage instance deallocated by a call to the **`free`** or **`realloc`** function is used (7.22.3).

— 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`** (7.22.3.3, 7.22.3.5).

— The value of the object allocated by the **`malloc`** function is used (7.22.3.4).

— The values of any bytes in a new object allocated by the **`realloc`** function beyond the size of the old object are used (7.22.3.5).

— The program calls the **`exit`** or **`quick_exit`** function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

— 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 (7.22.4.4, 7.22.4.7).

— The string set up by the **`getenv`** or **`strerror`** function is modified by the program (7.22.4.6, 7.24.6.2).

— A signal is raised while the **`quick_exit`** function is executing (7.22.4.7).

— A command is executed through the **`system`** function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

— 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 (6.10.8.2).

J.3.5 Integers

1 — Any extended integer types that exist in the implementation (6.2.5).

— Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a ~~trap~~ non-value representation or an ordinary value (6.2.6.2).

— 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).

— 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 (7.22.1.3, 7.29.4.1.1).

— Whether or not the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, or **wcstold** function sets **errno** to **ERANGE** when underflow occurs (7.22.1.3, 7.29.4.1.1).

— 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 (7.22.3).

— 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 (7.22.4.1, 7.22.4.5).

— The termination status returned to the host environment by the **abort**, **exit**, **_Exit**, or **quick_exit** function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

— The value returned by the **system** function when its argument is not a null pointer (7.22.4.8).

— The range and precision of times representable in **clock_t** and **time_t** (7.27).

K.3.5.3.9 The `vfscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfscanf_s(FILE * restrict stream,
      const char * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vfscanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`vfscanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`vfscanf_s`** function is equivalent to **`fscanf_s`**, 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_s`** function does not invoke the **`va_end`** macro.[429)]

Returns

5 The **`vfscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **`vfscanf_s`** 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.

K.3.5.3.10 The `vprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vprintf_s(const char * restrict format,
      va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. The `%n` specifier[430)] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **`vprintf_s`** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`vprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`vprintf_s`** function is equivalent to the **`vprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`vprintf_s`** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

[429)] As the functions **`vfprintf_s`**, **`vfscanf_s`**, **`vprintf_s`**, **`vscanf_s`**, **`vsnprintf_s`**, **`vsprintf_s`**, and **`vsscanf_s`** invoke the **`va_arg`** macro, the ~~value~~ representation of `arg` after the return is indeterminate.

[430)] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

K.3.5.3.11 The vscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vscanf_s(const char * restrict format,
      va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vscanf_s** function is equivalent to **scanf_s**, 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_s** function does not invoke the **va_end** macro.[431)]

Returns

5 The **vscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vscanf_s** 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.

K.3.5.3.12 The vsnprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsnprintf_s(char * restrict s, rsize_t n,
      const char * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than **RSIZE_MAX**. The `%n` specifier[432)] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **vsnprintf_s** corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than **RSIZE_MAX**, then the **vsnprintf_s** function sets `s[0]` to the null character.

Description

4 The **vsnprintf_s** function is equivalent to the **vsnprintf** function except for the explicit runtime-constraints listed above.

5 The **vsnprintf_s** function, unlike **vsprintf_s**, will truncate the result to fit within the array pointed to by `s`.

[431)] As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the ~~value~~ representation of `arg` after the return is indeterminate.

[432)] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

Description

4 The **vsscanf_s** function is equivalent to **sscanf_s**, 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_s** function does not invoke the **va_end** macro.[434)]

Returns

5 The **vsscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vscanf_s** 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.

K.3.5.4 Character input/output functions

K.3.5.4.1 The gets_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
char *gets_s(char *s, rsize_t n);
```

Runtime-constraints

2 `s` shall not be a null pointer. `n` shall neither be equal to zero nor be greater than **RSIZE_MAX**. A new-line character, end-of-file, or read error shall occur within reading `n-1` characters from **stdin**.[435)]

3 If there is a runtime-constraint violation, characters are read and discarded from **stdin** until a new-line character is read, or end-of-file or a read error occurs, and if `s` is not a null pointer, `s[0]` is set to the null character.

Description

4 The **gets_s** function reads at most one less than the number of characters specified by `n` from the stream pointed to by **stdin**, into the array pointed to by `s`. No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

5 If end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then `s[0]` is set to the null character, and the other elements of `s` take unspecified values.

Recommended practice

6 The **fgets** function allows properly-written programs to safely process input lines too long to store in the result array. In general this requires that callers of **fgets** pay attention to the presence or absence of a new-line character in the result array. Consider using **fgets** (along with any needed processing based on new-line characters) instead of **gets_s**.

Returns

7 The **gets_s** function returns `s` if successful. If there was a runtime-constraint violation, or if end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then a null pointer is returned.

[434)] As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the ~~value~~ representation of `arg` after the return is indeterminate.

[435)] The **gets_s** function, unlike the historical **gets** function, makes it a runtime-constraint violation for a line of input to overflow the buffer to store it. Unlike the **fgets** function, **gets_s** maintains a one-to-one relationship between input lines and successful calls to **gets_s**. Programs that use **gets** expect such a relationship.

Returns

7 The **qsort_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.6.4 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 function is placed into its initial conversion 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 set the int pointed to by their status argument to a nonzero value if encodings have state dependency, and zero otherwise.[444] Changing the **LC_CTYPE** category causes the internal object describing the conversion state of these functions to ~~be indeterminate~~ have an indeterminate representation.

K.3.6.4.1 The wctomb_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t wctomb_s(int * restrict status,
      char * restrict s,
      rsize_t smax,
      wchar_t wc);
```

Runtime-constraints

2 Let n denote the number of bytes needed to represent the multibyte character corresponding to the wide character given by wc (including any shift sequences).

3 If s is not a null pointer, then smax shall not be less than n, and smax shall not be greater than **RSIZE_MAX**. If s is a null pointer, then smax shall equal zero.

4 If there is a runtime-constraint violation, **wctomb_s** does not modify the int pointed to by status, and if s is not a null pointer, no more than smax elements in the array pointed to by s will be accessed.

Description

5 The **wctomb_s** function determines n and stores the multibyte character representation of wc in the array whose first element is pointed to by s (if s is not a null pointer). The number of characters stored never exceeds **MB_CUR_MAX** or smax. 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.

6 The implementation shall behave as if no library function calls the **wctomb_s** function.

7 If s is a null pointer, the **wctomb_s** function stores into the int pointed to by status a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings.

8 If s is not a null pointer, the **wctomb_s** function stores into the int pointed to by status either n or -1 if wc, respectively, does or does not correspond to a valid multibyte character.

9 In no case will the int pointed to by status be set to a value greater than the **MB_CUR_MAX** macro.

Returns

10 The **wctomb_s** function returns zero if successful, and a nonzero value if there was a runtime-constraint violation or wc did not correspond to a valid multibyte character.

[444] 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.

3 If there is a runtime-constraint violation, the **vfwprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vfwprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **vfwprintf_s** function is equivalent to the **vfwprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vfwprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.7 The vfwscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf_s(FILE * restrict stream,
      const wchar_t * restrict format, va_list arg);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vfwscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vfwscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vfwscanf_s** function is equivalent to **fwscanf_s**, 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_s** function does not invoke the **va_end** macro.[466)]

Returns

5 The **vfwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vfwscanf_s** 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.

K.3.9.1.8 The vsnwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vsnwprintf_s(wchar_t * restrict s,
      rsize_t n,
      const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than **RSIZE_MAX/sizeof(wchar_t)**. The `%n` specifier[467)] (modified or not by flags, field width, or preci-

[466)] As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the ~~value~~ representation of `arg` after the return is indeterminate.

[467)] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was

K.3.9.1.10 The vswscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswscanf_s(const wchar_t * restrict s,
      const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vswscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vswscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vswscanf_s** function is equivalent to **swscanf_s**, 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_s** function does not invoke the **va_end** macro.[469)]

Returns

5 The **vswscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vswscanf_s** 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.

K.3.9.1.11 The vwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwprintf_s(const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. The `%n` specifier[470)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to **vwprintf_s** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vwprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vwprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **vwprintf_s** function is equivalent to the **vwprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vwprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

[469)] As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the ~~value~~ representation of `arg` after the return is indeterminate.

[470)] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

K.3.9.1.12 The vwscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwscanf_s(const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 format shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vwscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vwscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vwscanf_s** function is equivalent to **wscanf_s**, 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_s** function does not invoke the **va_end** macro.[471)]

Returns

5 The **vwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vwscanf_s** 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.

K.3.9.1.13 The wprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wprintf_s(const wchar_t * restrict format, ...);
```

Runtime-constraints

2 format shall not be a null pointer. The %n specifier[472)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by format. Any argument to **wprintf_s** corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **wprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **wprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **wprintf_s** function is equivalent to the **wprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **wprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

[471)] As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the ~~value~~ representation of arg after the return is indeterminate.

[472)] It is not a runtime-constraint violation for the wide characters %n to appear in sequence in the wide string pointed at by format when those wide characters are not a interpreted as a %n specifier. For example, if the entire format string was L"%%n".

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.[488)]

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~~unspecified values, and the representation of objects that are written to might become indeterminate.

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 (3.21.5), 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).

[488)] Implementations that do not define **__STDC_ANALYZABLE__** are not required to conform to these specifications.

Bibliography

[1] David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. C memory object and value semantics: the space of de facto and ISO standards. http://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf (a revison of ISO SC22 WG14 N2013), March 2016.

[2] Clive D. W. Feather. Indeterminate values and identical representations (DR260). Technical report, September 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

[3] Jens Gustedt. Pointers and integer types. Research Report N2889, ISO JCT1/SC22/WG14, January 2022. URL https://hal.inria.fr/hal-03363711.

[4] Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B F Gomes, and Martin Uecker. Moving to a provenance-aware memory object model for C: proposal for C2x. Technical Report N2362, ISO JCT1/SC22/WG14, April 2019. URL https://hal.inria.fr/hal-02089889.

[5] Krebbers and Wiedijk. N1637: Subtleties of the ANSI/ISO C standard, September 2012. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf.

[6] Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015.

[7] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code with twin memory allocation. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, November 4-9, 2018*. ACM, 2018.

[8] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (Santa Barbara)*, June 2016. URL http://www.cl.cam.ac.uk/users/pes20/cerberus/pldi16.pdf. PLDI 2016 Distinguished Paper award.

[9] Kayvan Memarian, Victor B. F. Gomes, Kyndylan Nienhuis, Justus Matthiesen, James Lingard, Stella Lau, and Peter Sewell. Cerberus tool for exploring the semantics of the c programming language., 2016–2022. http://cerberus.cl.cam.ac.uk/cerberus.

[10] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. In ***POPL 2019**: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. doi: 10.1145/3290380. Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO WG14 N2311, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2311.pdf.

[11] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014. doi: 10.1145/2628136.2628143. URL http://doi.acm.org/10.1145/2628136.2628143.

[12] Peter Sewell, Kayvan Memarian, Victor B F Gomes, Jens Gustedt, and Martin Uecker. C provenance semantics: examples. Technical Report N2363, ISO JCT1/SC22/WG14, April 2019. URL https://hal.inria.fr/hal-02089907.

[13] Martin Uecker and Jens Gustedt. Indeterminate Values and Trap Representations. Research Report 2772, ISO TC1/SC22/WG14, July 2021. URL https://hal.inria.fr/hal-03408023.

