**DRAFT
Technical
Specification**

ISO/DIS TS 25007

Programming Languages — C — C Extensions to support generalized function calls

ISO/ TC22/SC22
Secretariat: JISC

Voting begins on: n/a

Voting terminates on: n/a

Reference Number
ISO/DIS TS 25007 : Working Draft N3582

Project Editor: Alexandria Celeste
(aceleste@perforce.com)

Content

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).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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 Technical Specification aims to lay out common practice guidelines for features related to extended function call support.

Two main feature areas are covered: extended constant expressions, and proper tail calls. These extension features are of interest to application developers who want to take advantage of language-level abstractions beyond the usual restrictions of translation and execution phases. In both cases the concept of a function call is generalized to allow users to write readable code that uses C features in an intuitive way, in places where it would not be accessible in the core language.

The first feature area, which generalizes `constexpr` to functions and arrays, is of relevance to the interop between C and C++, as it proposes to expand the overlap between these two languages and thus improve header compatibility and the usefulness of header definitions. This builds upon features that were accepted for standardization in C23.

The second feature area, tail-call elimination, is also of special relevance to the wider language community, by aiming to improve interop between C and other languages already having a more generalized concept of function calls that allows for in-place replacement.

In both cases the intent is to codify common and unified practice.

1 Scope

This Technical Specification specifies a series of extensions of the programming language C, specified by the international standard ISO/IEC 9899:2024.

Each clause in this Technical Specification deals with a specific topic. The first sub-clauses of clauses 4 through 7 contain a technical description of the features of the topic. These sub-clauses provide an overview but do not contain all the fine details. The last sub-clause of each clause contains the editorial changes to the standard necessary to fully specify the topic in the standard, and thereby provides a complete definition.

2 References

For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 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:

ISO/IEC 9899:2024, Programming languages — C
TS 25755, Programming Languages — C — defer, a mechanism for general purpose, lexical scope-based undo

2.2 Informative References

The following documents are important for understanding the concepts underpinning the features described in this document:

ISO/IEC 14882:2011, Programming languages — C++
ISO/IEC 14882:2014, Programming languages — C++
ISO/IEC 14882:2020, Programming languages — C++
Revised^7 Report on the Algorithmic Language Scheme: https://small.r7rs.org/attachment/r7rs.pdf
Compiling Higher-Order Languages into Full Tail-Recursive Portable C (Feeley, Miller, Rozas, Wilson): https://www-labs.iro.umontreal.ca/~feeley/papers/FeeleyMillerRozasWilsonDIRO1078.pdf
Struct expressions (Rust): https://doc.rust-lang.org/reference/expressions/struct-expr.html
`musttail` in Clang: https://clang.llvm.org/docs/AttributeReference.html#musttail
`musttail` in GCC: https://gcc.gnu.org/onlinedocs/gcc/Statement-Attributes.html
`recur` in Clojure: https://clojuredocs.org/clojure.core/recur

2.3 Additional References

Further historical context and the development of this document is attested in these historical working group N-documents:

N2892 “Basic lambdas for C”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2892.pdf
N2917 “The `constexpr` specifier, v2”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2917.pdf
N2920 “Tail-call elimination, v2”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2920.pdf
N3018 “The `constexpr` specifier for object definitions, v7”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3018.htm
N3199 “Improved `__attribute__((cleanup(…)))` through **`defer`**”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3199.htm
N2361 “Out-of-band bit for exceptional `return` and `errno` replacement”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2361.pdf
N3311 “Array subscripting without decay”: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3311.htm

3 Conformance

This Technical Specification presents in separate clauses specifications for two, in principle independent, sets of functionality, which are primarily described by Clause 5: Tail-call elimination; and Clause 6: Extended Constant Expressions.

These are supported by explanatory functionality in Clause 4, which defines a small number of utility features only so that the concepts can be used by the descriptions in the subsequent clauses; and additional extended functionality in Clause 7, which separates out some features related to Clause 6 in order to simplify conformance decisions by implementers.

The features defined by Clause 4 are intended to be explanatory and they do not need to be integrated or adopted in order to implement subsequent clauses. The features defined by Clause 7 build upon the features defined by Clause 6 and do not need to be adopted in order to implement Clause 6 alone.

As this is a Technical Specification, there are no conformance requirements, and implementers are free to select those specifications that they need. However, if functionality is implemented from one of the feature clauses, implementers are strongly encouraged to implement that clause in full, and not just a part of it. The purpose of this Specification being to codify common practice, implementers are strongly encouraged to document any deviations or omissions in order to establish which practices are of more and less interest to the target audience.

If, at a later stage, a decision is taken to incorporate some or all of the text of this Technical Specification into the C standard, then at that moment the conformance issues with respect to (parts of) this text need to be addressed (conformance with respect to freestanding implementations etc.).

3.1 Terms and definitions

For the purposes of this document, the terms and definitions of ISO/IEC 9899:2024 apply.

4 Explanatory Functionality

This clause describes four sub-features that are not required to be implemented in order to provide the main functionality in clauses 5 through 7, but which simplify the non-normative descriptions therein substantially by being available to use in the semantic descriptions.

Implementers are neither encouraged to provide, nor discouraged from providing, these features as functionality in their own right; they are not the focus of this Technical Specification and are not necessarily completely specified.

Full normative text is not provided for these features.

4.1 Calling convention specifier

On many target platforms, functions with the same Standard type may have more than one incompatible ABI. The calling convention is a property of a function implementation which is completely outside the scope of the Standard, because it refers to the exact details of where and how arguments and other things needed for a call are passed around on the concrete machine.

Two functions with different calling conventions can have the same Standard type, but absolutely must not be treated as having compatible types by the implementation, as trying to call one through a pointer expecting the other would end up doing something along the lines of putting values in the wrong registers for the subsequent instructions.

For the purpose of describing functions that have different types despite having the same Standard type, we add a new parameterized "function qualifier" `_Call_as`.

Unlike the "object qualifiers" `const`, `volatile`, `restrict` and `atomic`, the `_Call_as` qualifier modifies a function type. The qualifier accepts an integer constant expression as a single mandatory argument. When `_Call_as` syntactically qualifies a function return type, it is adjusted to qualify the nearest syntactically enclosing function type.

```
_Call_as (1) void f1 (void);    // f1 is qualified by adjustment
void (_Call_as (1) f1) (void);  // compatible declaration, directly qualifying the identifier

typeof (void (void)) _Call_as (2) * f2 (void);  // the qualifier modifies the pointed-to type
                                                // and is not adjusted, so f2 is not qualified
                                                // but its return value is

typeof (_Call_as (2) void (void)) * f2 (void);  // compatible declaration with the above
                                                // the return type of the typename is modified
                                                // so the surrounding function type is adjusted

typedef void Func (void);
Func _Call_as (3) * fp;    // qualifying a function type
```

The combination of `_Call_as` with an integer constant *value* is called a *convention qualifier*. Two different constant values produce different convention qualifiers.

Following existing qualifier rules, two types qualified with different convention qualifiers are not compatible. An unqualified function type is compatible with exactly one implementation-defined convention qualifier. Consequently there is no implicit conversion from an unqualified to any convention-qualified function type except the sole implementation-defined compatible qualifier, because there is no "convention-unqualified" function type. This avoids a previously encountered problem (described in N2361) where functions could potentially be implicitly converted to signatures with an incompatible calling convention but compatible language type.

It is assumed that an implementation will provide predefined constants to represent the calling conventions it supports. An implementation should reject convention qualifiers it does not support.

```
// constexpr auto SYSV_ABI = 1;
// constexpr auto MS_ABI = 2;

typedef void Func (void);

typedef Func _Call_as (MS_ABI) MSFunc;
typedef Func _Call_as (SYSV_ABI) SVFunc;

MSFunc * f1 = some_function;
SVFunc * f2 = f1; // ERROR: types are not compatible

typedef void VarFunc (...);
VarFunc * f3 = f2; // ERROR obviously, (void) and (...) are different Standard types
```

There is no particular guarantee that two incompatible function types qualified with the same convention qualifier actually use a similar calling convention at the assembly level.

Convention qualifiers are a special case of user-defined qualifiers. It does not make sense for a convention qualifier to be implicitly convertible on-or-off, or to have multiple qualifiers at once.

4.2 Functional update expression

Functional update is a useful feature predominantly found in functional languages that creates a value of a given type with the same contents as an existing value, with some explicit changes.

This is useful for structures because it allows setting only members of interest without needing to know about all other members that exist. Code that explicitly initializes all members can become out of date in the event of a modification of the structure definition, and is also much more verbose.

Functional update syntax exists in Rust. It can be adopted for C by modifying the initializer syntax slightly to allow for a leading “default object” initializer, before the individual member initializers:

```
struct Point3D {
    float x, y, z;
};

struct Point3D p1 = { .x = 1, .y = 2, .z = 3 };

// move a second point in y only
struct Point3D p2 = { p1; .y = 6 };
auto p3 = (struct Point3D){ p1; .y = 6 }; // same result
```

The semicolon is not otherwise used in *initializer-list* syntax and is more C-like than the range operator preferred by Rust.

This is resilient against changes to the definition of a type, or even generic to a type entirely:

```
#define TranslateY(val, by) (typeof(val)){ val; .y = (val).y + by }

struct Point2D ...
struct Point3D ... // all have a y
struct Point4D ... // don't need to know about x, z or w
```

The expression in the “default object” position is not value-converted; if it designates an array, the elements of the array are used to pre-initialize the object.

If the initializer list has exactly one element, array element designators within the *designator-list* may use non-constant expressions to designate an element. The expression shall evaluate to a non-negative value less than the length of the array object or sub-object.

The type produced by the `typeof_unqual` operator applied to the type of the “default object” expression shall be compatible with the type produced by `typeof_unqual` applied to the type of the object being initialized.

4.3 Function literal

A function literal is an expression that evaluates to the value of a function defined inline in the same expression where it is used. The function has no user-observable name, and the definition is provided immediately at the single point of usage. Since the expression will decay to a function pointer because of value conversion, it can be assigned to a named function pointer or called in place, allowing it to be used in an expression exactly like any named function’s identifier.

The syntax for a function literal is similar to a function definition, but instead of the return type and identifier of a function declarator, the expression begins with `[]`, followed by the parameter list as usual and then the function body. The parameter list is optional and is implicitly empty if not provided.

```
void foo (int x) {
    return x + [] (int y) {
        return y + 5;
    } (6);
}

// is exactly equivalent to

static int @foo__unnamed1 (int y) { // the name cannot be expressed in user code
    return y + 5;
}

void foo (int x) {
    return x + @foo__unnamed1 (6);
}
```

The function definition’s body forms a block scope, but it does not have access to objects with automatic storage duration in the surrounding block scope. Type definitions and objects with static or thread-local storage duration in the surrounding block scope *are* visible within the function literal, as though they had been defined at file scope with a unique identifier:

```
int bar (int x) {
    typedef int Int;
    struct Local { Int z; };

    return [] (Int y) {
        struct Local l = { .z = y + 1 };
        return l;
    } (x).z;
}

// is exactly equivalent to

typedef int @bar__Int;
struct @bar__Local {
    @bar__Int z;
};

static @bar__Int @bar__unnamed1 (@bar__Int y) {
    struct @bar__Local l = { .z = y + 1 };
    return l;
}

int bar (int x) {
    return @bar__unnamed1 (x).z;
}
```

Variably-modified types in the surrounding block scope are not visible in the function literal. Names in the surrounding block scope that shadow declarations at file scope prevent those file scope declarations from being visible to the function literal even if it cannot make use of them.

The return type of the function literal is deduced from any `return` or `return goto` statements inside its definition. If there are no such statements, the return type is `void`. Otherwise, the return type is the type after value conversion of the operand expression to the first return statement appearing in lexical order in the function body. If the return statement has no expression, the return type is `void`. The type of the expression of any subsequent return statements shall be compatible with the deduced type.

Any function literal can be trivially rewritten out-of-line as a normal C function at file scope, though the types and objects it references may need to move to file scope as well.

A function literal implicitly has internal linkage and is implicitly an inline function. A function literal that meets the constraints for the `constexpr` specifier is implicitly a constexpr function.

Function literals are a special simplified case of what are called "lambdas" in other languages. For the purposes of illustrating constexpr functions, only the inline definition and static scope features are useful.

4.4 `_Recur` pointer

`_Recur` is a predefined identifier that behaves like any function designator, except that it implicitly refers to the definition of the function containing the expression in which it appears.

This is useful for expressing recursion without knowing the name of the function being invoked recursively, which may be because the function has no user-accessible name (see 4.3), or because the recursion is some generic algorithm-building operation invoked from a macro intended for reuse.

The type of `_Recur` is the same as the type of the corresponding function designator before value conversion.

```
int bar (int);

int foo (int x) {
    // assert that bar is compatible with foo
    typeof (_Recur) * pbar = bar;
    return pbar (x);
}

int fib (int n) {
    // anonymously define a recursive function
    auto aux = [] (int i, int a, int b) {
        if (i == 0)                     // this can't be a ternary:
            return a;                   //   need to deduce the type from return
        return _Recur (i - 1, b, a + b); //   so the type is known before _Recur
    };
    return aux (n, 0, 1);
}
```

The name is taken from a related feature in the Clojure language, `recur`.

5 Tail-Call Elimination

5.1 Introduction

C's function activation records implicitly form a stack, regardless of the underlying implementation of function calls or the depth of calls which the platform can support.

Keeping all activation records suspended but alive until their callees have returned prevents programs from transferring control directly when a function's work is already done. This is not a huge obstacle for hand-written C code, but is a major barrier for languages that compile to C (which cannot then generally represent their function calls as native C function calls, instead resorting to complicated non-native function representations (Feeley *et al.*), or for interop with languages that do support this feature.

This Specification adds the ability to explicitly, directly transfer control to another function without returning, as an alternative to nested function calls.

This feature was previously discussed during the January 2022 meeting of WG14, and the meeting established some starting directions for the feature set:

— there was not consensus to add the feature to C23;
— there was direction to continue gathering implementation experience;
— there was direction to establish what impact (if any) the feature has on existing ABIs.

Since the principal implementation obstacles are in ABI and calling convention, both of which are currently out of scope for the Standard and are not defined by C23, the preference of the Committee was that a tool should be able to reject a tail call for any implementation-defined reason. This also allows a tool to simply not support tail calls at all, so long as it emits a diagnostic and does not silently compile the `return goto` statement as though it was a simple `return` statement.

The rationale and prior art for *tail-call* elimination on return statements is explained and discussed in WG14 document N2920.

A deeper understanding of *tail-call* elimination in general can be obtained from the specification of the Scheme programming language (R7RS), which mandates its support.

The feature specified here is fundamentally similar to the Clang `musttail` feature, with appropriate modifications to fit into the Standard language and alongside the other core feature in this Specification. A compatible feature was made available in GCC in 2024, after originally being implemented in 2016.

5.2 Overview of *tail-call* elimination in C

For the purpose of this Specification, a *tail-call* is a call to a function that consists of the entire value operand to a `return goto` statement (or one of the related forms below).

Although most of the research on tail calling in C functions has been into identifying implicit *tail-calls* for the purposes of optimization, this Specification is only concerned with explicit *tail-calls* requested by means of the `return goto` statement. *Tail-calls* eliminated explicitly in this way are **not an optimization**, and directly affect the program semantics in potentially observable ways. For this reason a new statement kind, the `return goto` sub-type of the `return` statement, is needed, as an attribute cannot impose the required change in behavior of a normal `return` statement.

Entry to a *tail-call* ends the lifetime of the calling function's activation record, and must free any resources created by that function call. This moves the end of lifetime of automatic objects with block scope defined within the caller to immediately before entry to the function being called in tail position. This does not introduce any new undefined behaviors directly, but does mean that pointers to objects in the calling activation record become invalid before the *tail-call* itself.

The expression in tail position must not require any additional work to be done after the function returns within the body of the caller, because the caller will not be re-entered. This means that as well as being a syntactic function call, the operand to `return goto` must have a type exactly matching the return type of the calling function, without needing to undergo any implicit conversions. Any implicit conversion, even if it would compile as a no-op, renders the operand an invalid *tail-call*.

In order to fully enforce this, the function called in tail position must have identical type to the callee. This ensures both that the return value does not require any conversion, and also that argument passing space is available and calling convention (if relevant) is maintained.

The implication of the requirement that all resources allocated by the immediate caller are freed right before entry to the callee is that any sequence of *tail-calls* will never overflow the implementation's stack or otherwise cause resource exhaustion because of the calls themselves. An infinite sequence of *tail-calls* is analogous to an infinite loop – although it may do other work that uses resources, it does not use resources itself by iterating.

5.3 Calling conventions and ABI

No new ABI is introduced by this feature. Although the function called from tail-position replaces its syntactic caller, entry re-uses the same argument and return space that was set up for that caller. Therefore, whether a function ends in a tail call is not generally observable from "outside". Because the called function must match the caller's setup exactly, it must by definition have the same ABI and is also callable from any non-tail position, making its potential status as a *tail-call* completely private to the call context and not observable from its definition or declaration at all. The caller and the callee may require a different static size for their activation records, but there does not appear to be a target (where the stack discipline is actually implemented in a re-entrant way) where this would be problematic (on targets with re-entrant stack disciplines, the top of the frame is always necessarily set within the function body because it is not exposed by the type anyway; on those with non-contiguous stacks a different object is used; etc.).

Any function which can be invoked by a *tail-call* may also be invoked by a function call in non-tail position. Any function which terminates in a *tail-call* may also terminate in a conventional `return` statement along a different conditional branch.

The principal ABI impact is on functions that currently use workarounds such as trampolining in order to emulate proper *tail-calls*. These functions incorporate the emulation into their ABI at present (using signatures that allow for "return to next" by some means). It is assumed that users of such functions will mostly *actively want* their ABI to change and become seamless, once that becomes possible. However, explicit trampolining will continue to work unchanged and will not be impacted by the addition of this feature unless it is intentionally picked up by the user.

It is not generally possible to perform a *tail-call* between functions that have different calling conventions. Although this is outside the scope of the Standard, by placing constraints on the type of the callee function as a whole (rather than expressing separate constraints against its return type and arguments), calling conventions **should** implicitly be constrained as well, because an implementation already has to treat them as having distinct types, in order to emit correct code (i.e. pointers to two functions with different conventions should not be stored in the same pointer variable, a high-quality compiler should make a type-based distinction). Because of this, specifics of the calling convention ("caller cleanup" vs "callee cleanup", and so on) are not considered; so long as the constraint that the complete type of the function called in tail position matches **exactly** is respected, these are expected to be respected by the callee. In the description below (non-normative) calling convention is treated as part of the function type.

5.3.1 Rejecting tail calls

During the January 2022 WG14 meeting it was pointed out that some targets use different calling conventions for the same function depending on whether the call is a "near call" (within the same TU) or a "far call" (callee is in another TU). If a function ends with a *tail-call* to a "far" target, all calls to the calling function themselves must become "far calls" so that it can be perfectly replaced in-place. This means that a TU making *tail-calls* to other functions in other TUs would need to be recompiled to ensure it contains no "near calls" to callers.

Since all local calls would be recompiled when the function was rewritten to use `return goto`, it was not clear whether this would present an issue in practice. An implementation is encouraged to tag such functions during compilation to force them to always use a single calling convention. This way, so long as the internal call also always uses the same convention, the *tail-call* is preserved and the detail about how the function terminates remains semantically private.

During the October 2024 WG14 meeting it was pointed out that some targets support dynamically changing the prologue or epilogue of a function without changing its calling convention. Since this is a runtime feature, *all* functions where this is potentially a feature should be assumed to not be eligible for tail-calling. The solution in this case is that the compiler for such a platform would have to reject most calls as impossible to eliminate, except for cases where it has full inlining information.

In general the feedback from the Committee was that implementations should be allowed to reject a requested tail call for any implementation-defined reason, including that the implementation is not able to emit tail calls in general. So long as the implementation **rejects** the program rather than accepting it with conventional return semantics (which would result in different resource usage and likely overflow), this is a safe implementation-defined behavior to add, and handily covers all cases where the call would be difficult to translate.

Both GCC and Clang have encountered areas where translating a `musttail` call is not possible, and both tools reject a program using the attribute when such a call is correctly identified rather than attempting to ignore it.

An implementation which rejects all `return goto` statements is **therefore conforming** so long as it at least parses and recognizes the syntax.

5.4 Informative semantic description

This clause introduces one new statement, with no new internal syntax but with some Constraints; adjusts the lifetime/duration rules for objects around the semantics of the new statement; and adds one new permissible use for `void`-typed expressions.

The new statement added is the `return goto` statement, which is closely related to the `return` statement in that it marks the end of execution of the current function. However, while the `return` statement evaluates its operand (or, has no operand) within the current execution function's active context, the `return goto` statement terminates execution of the current function *before* fully completing evaluation of its operand expression.

A `return goto` statement can accept three main kinds of expression:

— a syntactic atom (literal, lexical constant, identifier); note that “lexical constant” here refers to syntactically atomic constants (also commonly called “literals”, though the C23 Standard does not use this term for numbers), and not to *constant-expressions* in general;
— a function call expression, upon which there are additional constraints;
— a conditional (“ternary”) expression, selecting between either of the above, or a nested ternary with the same restriction.

The atom/ternary kinds are not expected to be the common case and are really only provided for compatibility with the unrelated `constexpr`-function feature in Clause 6 (without allowing these, it would be impossible to use `return goto` to define a recursive constexpr function because the recursion would be impossible to terminate; “normal” non-constexpr functions are able to just use an `if` statement).

The following examples are therefore valid:

```
int a (int, int);

int b (int x, int y) {
    return goto a (y, x); // OK - function call to an identical signature
}
int c (int x, int y) {
    return goto x && y    // OK - conditional, selecting between...
         ? a (y, x)       // ... function call to an identical signature
         : x;             // ... syntactic atom
}
int d (int x, int y) {
    // OK - conditional, selecting between...
    return goto x ? (a (y, x)) // ...function call and another conditional
              : y ? (b (y, x)) // ...function call and a syntactic atom
              :     (y);
}
int e (void) {
    return goto 6.5f;  // useless, but OK - syntactic atom
}                      // (the implicit conversion here doesn't matter)
```

while the following are not:

```
int a (int, int);

int b (int x, int y) {
    return goto a (y, y) + x;  // not a function call (+ happens afterwards)
}
int c (int x, int y) {
    return goto x && y
         ? a (y, x)            // this branch is OK...
         : x + y;              // ...but this doesn't obey the Constraints
}
int d (int x, int y) {
    return goto x == 0
              ? a (y, 0)       // this branch is OK...
              : a (y, 1) + x;  // but this wouldn't tail-call
}
int e (int x, int y) {
    return goto (long)a(x, y); // even a NOP-cast is still an operation
}
```

The function-call operand is the primary use case and has additional constraints and semantics:

— the called function must have a compatible signature with the containing/“caller” function;
— the called function must not be variadic;
— the called function must not have any implementation-defined features that would prevent its use as the target of a *tail call*.

As a consequence of this last point, implementations are granted broad freedom to reject a `return goto` statement for "any other reason", which does arguably mean that there is no strict requirement to support the statement's semantics at all. Implementations are *not* allowed to implement the statement as a simple `return` statement, as this would have incorrect semantics. As a quality-of-implementation matter, they should ideally explain *why* any given statement is rejected. "The target platform does not generally support tail calls" is an acceptable reason.

The following examples are therefore valid:

```
float a (float x, float y);
_Call_as(CDECL) int b (int x, int y);

float f (long i, long j) {
    return goto a (i + j, j + 2.0f); // OK - identical signature, args OK
}

typeof (int (int, int)) _Call_as(CDECL) * getFunc (void);

_Call_as(CDECL) int g (int x, int y) {
    return goto getFunc () (y, x);   // OK - identical signature including ABI
}
```

while the following are not:

```
int h (int i, int j) {
    return goto a (i + j, j + 2.0f); // implicit conversion caused by
}                                    // invalid signature

_Call_as (STDCALL) int k (int x, int y) {
    return goto getFunc () (y, x);   // signature is not identical in
}                                    // non-Standard elements

// implementation ABI that doesn't support direct jumps
_Call_as(GPU) float c (float r, float g, float b);

_Call_as(GPU) float grey (float r, float g, float b) {
    return goto c (r * 0.3f    // implementation rejects this
                 , g * 0.59f   // for "other reasons"
                 , b * 0.11f);
}
```

The order of operations for the `return goto` statement when its operand is a function call is as follows:

1. the operands to the function-call expression (the *postfix-expression* evaluating to a function pointer to the target of the call, and the argument values) are evaluated and value-converted, in the appropriate sequence (i.e. no different from the usual sequencing rules). These subexpressions can be any valid operands to a normal function call expression, and are not themselves subject to any of the Constraints associated with `return goto` - the *postfix-expression* can be a simple identifier directly designating a function, or any arbitrarily complex expression producing a function pointer; and the argument subexpressions can make free use of any object with block lifetime *while they are being evaluated*, for instance passing the address of a local to a nested function call within the argument.
2. execution of the caller is terminated and the lifetime of all objects with block scope ends. Consequently any references to objects with block scope in the caller's activation frame are now invalid.
3. the argument values, which exist in abstract-machine "value space" after evaluation in step 1 and are not themselves objects with lifetime, are copied to the implementation-defined storage for parameter objects. This storage is generally expected to reuse the parameter storage provided to the caller (although a valid implementation might e.g. also have some kind of ring-buffer structure to the activation records, so no guarantee is made about the addresses of parameters).

4. execution uses a *direct jump* of some implementation-defined kind to go to the start of the function designated in the call expression. Whatever the nature of this jump, it **replaces** the caller in the implicit execution stack, and **does not** form a stack on top of it.
5. execution of the callee proceeds (which might involve additional "nested" tail calls).
6. the callee's return value is passed directly to the receiver expecting the *caller's* return value.

No amount of "nesting" of tail calls in step 5 will ever use more than just the one stack frame.

The following examples are therefore valid:

```
int f (int x, int * prev) {
    static int y = 6;
    if (x > 10) {
        return goto f (y, &y); // OK - lifetime of y is the whole program
    }
    return goto f (x + y, prev); // OK - (x + y) is value-converted and not lost
}

int h (int, int *);

int g (int x, int * y) {
    return goto h (h (*y, &x)  // OK - call nested in argument
                 , y);         // can rely on usual lifetimes
}
```

while the following is not:

```
int h (int, int *);

int g2 (int x, int * y) {
    return goto h (*y, &x);  // &x is a dangling pointer
}
```

There is no guarantee that the stack frame for each function is the same size, and the direct jump may also need to adjust the "top of the stack" register if such a thing exists in the calling convention (this commonly happens inside the callee, after the jump/call, which is convenient).

In other words, a *tail call* is different from a normal function call in that:

— a normal function call says "pause me, get the result of this, continue".
— a *tail call* says "**replace me** to return the result of this as my result".

This consequentially creates the adjusted lifetime rules: if the execution of the caller *terminates* instead of suspends, the callee cannot make use by pointer of any object with block scope in the caller, because its storage is assumed to have been completely reused to make space for the callee. By changing the end of lifetime to "right before" entering the callee instead of "right after" fully evaluating the `return` expression, such references are made invalid before the callee is entered.

(In Scheme and other languages, *all* objects have dynamic lifetime semantics and the existence of a GC is assumed; this is completely alien to the way objects are allocated in C so this property is not preserved.)

(An implementation doesn't have to reuse the space *immediately*, so long as the persistence of old stack frames doesn't cause resource exhaustion; a hardened/debug implementation might keep some fixed number of frames alive for longer in order to detect invalid accesses, for instance. It is up to the implementation to determine whether this is useful for reasons outside of the semantics.)

Another consequence of the adjusted lifetime rule is that *absolutely nothing* may happen in the caller context after the callee has been entered. This means that the **`defer`** statement, or `__attribute__((cleanup))`, or anything like that, cannot be used to schedule any actions on the exit from a function via `return goto`, because there is nowhere and no-when for the action to run. Scheduling these actions *before* entry to the callee would be so counter-intuitive it is safer to just require the user not to combine these features in such a broken way.

If the implementation needs to add additional instructions to de-allocate a VLA at the end of its lifetime, this should ideally happen right before entry to the callee (there is no observable side effect of this action, so it can be scheduled. If this is impossible for the implementation to generate code for, then it would become an “implementation-defined reason” to reject the statement.

The `return goto` statement is also slightly relaxed in one respect from the `return` statement, which is that the overall operand expression may be `void`-typed, if and only if the containing function (and the callee) have a `void` return type. In this case no value is returned to the outer context but it is still possible to replace the caller directly to do some work involving side effects.

(This is not relevant to constexpr functions, so no special relaxation needs to be applied to the ternary operator or to identifiers in order to make advanced use of `void` values.)

5.5 Detailed changes to ISO/IEC 9899:2024

The modifications are ordered according to the clauses of ISO/IEC 9899:2024 to which they refer. If a clause of ISO/IEC 9899:2024 is not mentioned, no changes to that clause are needed. New clauses are indicated with (NEW CLAUSE), however resulting changes in the existing numbering are not indicated; the clause number *mm.nn***a** of a new clause indicates that this clause follows immediately clause *mm.nn* at the same level. Bolded text within an existing clause is new text.

Add a new section to the glossary in the appropriate position:

3.xx

tail call

function call executed as a direct jump from one function to a new function with an identical signature, ending the lifetime of the caller before the called function is entered and whose ultimate return value, if any, will be used as the return value of the caller

Note 1 to entry: a tail call may be a recursive call to the containing function, which is guaranteed to have the right signature.

Note 2 to entry: tail calls may invoke other functions via tail calls, with the ultimate return value being produced by a deeply-nested invocation.

Modify 6.2.4 "Storage durations of objects", paragraph 6, to clarify that some function calls do terminate execution:

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 **, unless entry to the call explicitly terminates execution of the current block with `return goto`**.)

Add a forward reference to (new section) 6.8.7.5.1 in 6.2.4:

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), **tail calls (6.8.7.5.1),** initialization (6.7.9), statements (6.8), effective type (6.5).

Modify the first sentence of 6.3.3.2 `void`:

The (nonexistent) value of a void expression (an expression that has type void) shall not be used in any way, **except as the operand of a `return goto` statement within a function with `void` return type,** and implicit or explicit conversions (except to `void`) shall not be applied to such an expression.

Add a new footnote to 6.5.3.3 "Function calls" paragraph 8, immediately after 92:

with respect to the execution of the called function. [92) **footnote)**]

footnote) if the called function is assuming direct control with the `return goto` statement, then no further operations in the caller will be evaluated after it is entered.

Modify 6.8.7.1 paragraph 1:

jump-statement:
  `goto` *identifier* ;
  `continue` ;
  `break` ;
  `return` *expression* $_{\text{opt}}$;
  `return goto` ***expression*** **;**

Add a new paragraph to 6.8.7.1 after paragraph 1:

The `return` and `return goto` statements are collectively called the *return statements*.

(Non-detailed) Change all instances of "`return` statement" to "return statement" (remove monospace) except where they specifically refer to the non-`goto` form.

(NEW CLAUSE) Add a new section, 6.8.7.5.1 "The `return goto` statement":

6.8.7.5.1 The `return goto` statement

Constraints

The expression shall be, ignoring any parenthesization and after generic selection, one of:

- a function call expression, henceforth *FC*, where the postfix-expression designating the function to call has a type compatible with the type of a pointer to the containing function; or
- a lexical constant, literal, or identifier, henceforth *LX*; or
- a conditional expression, henceforth *CX*, where the first operand may be any expression and the second and third operands are recursively subject to these constraints.

Neither the function called in the postfix-expression of *FC* nor the containing function shall have a parameter list that terminates with an ellipsis.

If the containing function is defined as returning `void`, the expression shall be a *void expression.*

(if the **`defer`** *statement specified by TS 25755 has been implemented)* The containing function shall not register any deferred operations to be executed on exit from any scope enclosing the `return goto` statement.

A conforming implementation may reject a `return goto` statement for additional implementation-defined reasons.[footnote)] An implementation that rejects a `return goto` statement shall not translate it as though it had been written as a simple `return` statement.

footnote) such as an incompatible calling convention that is not represented in the standard type, or even because the implementation does not support such direct jumps at all.

Semantics

A `return goto` statement is a special case of the `return` statement which terminates execution of the current function and then either passes direct control to the function specified in a function call sub-expression *FC* that evaluates its *operand value*, or returns the value of a primary sub-expression *LX*.

When the operand to a `return goto` statement is a conditional expression that is a valid instance of *CX*, the conditional expression is evaluated normally and the *operand value* of the statement becomes the selected result operand, recursively subject to these semantics.

When the operand is a lexical constant, literal, or identifier, it is an instance of *LX* and the behavior is identical to that of a simple `return` statement.

Otherwise the operand is a function call that is a valid instance of *FC*.

The value of *FC* will be returned to the caller's context as if by a simple `return` statement. The `return goto` statement differs from the `return` statement in that it terminates execution of the current function immediately after evaluating all operands to the function call expression, [footnote)] and before entering the called function itself. The called function will therefore return directly to the current function's caller (unless it also forwards control to a function using `return goto`, or does not return at all). This is called a *tail call.*

footnote) and after the implicit copy of any argument values to their associated parameter storage.

Because the execution of the current function is terminated, the lifetime of all objects local to the function with automatic storage duration ends immediately before the called function is entered. The statement shall not terminate the lifetime of any object with automatic storage duration for which the implementation would need to execute additional cleanup instructions.footnote)

footnote) any such instructions are an implementation detail that does not exist in the abstract machine, but may apply to variable-length arrays or implementation-defined types.

A function that has been terminated by the `return goto` statement does not continue to use resources.

NOTE: a `return goto` statement may appear anywhere a return statement with an expression may appear.

NOTE: if the address of a local object with automatic storage duration is passed to the called function, its lifetime will have ended before the called function begins executing and the resulting pointer cannot be used.

EXAMPLE 1 This example violates the constraint that the expression must be a direct call to a function returning the exact same type as the caller:

```
int foo (int, int);

int bar (int a, int b) {
  return goto foo (b, a) + 1; // WRONG: the +1 must be evaluated
}                             // after the result of foo()

float baz (int a, int b) {
  return goto foo (b, a);     // WRONG: the result of foo() is followed
}                             // by an implicit conversion
```

EXAMPLE 2 In this example the address of a local object with automatic storage duration is passed to a called function:

```
int foo (int * p); // uses p

int bar (int a) {
  return foo (&a); // OK, lifetime of a continues until foo() completes
}

int baz (int b) {
  return goto foo (&b); // WRONG: using the address of an object
}                       //         whose lifetime has ended

int * boo (int c) {
  return &c; // roughly analogous to the above
}
```

EXAMPLE 3 In this example, a function recurses endlessly but harmlessly because the recursive call consumes no additional resources:

```
int foo (int a, int b) {   // need space for locals...
  return goto foo (b, a);  // ...but that ends here
}
```

This class of function cannot overflow the program stack by itself.

EXAMPLE 4 In this example only one of the two calls to `foo()` is in the tail position:

```
int foo (int a, int b);

int bar (int a, int b) {
  return goto foo ( // will be evaluated after this caller's lifetime ends
      foo (a, b),   // will be evaluated within this caller's lifetime
      a + b
  );
}
```

The first nested call takes place before termination of the calling function and therefore must consider that its resources have not yet been released, exactly as for any other function call that is not in tail position.

EXAMPLE 5 In this example, the expression is a conditional expression:

```
    int foo (int a, int b);
    int qux (int a, int b);
    int toto (int a, int b, int c);

    int bar (int a, int b) {
      // this expression is a CX conditional,
      // whose second operand is a call expression,
      // and whose third operand is another conditional,
      // with a function call and an identifier as its own second and third operands
      // the structure of the first conditional operand doesn't matter
      return goto toto (a + b, a, b) ? foo (a, b)    // valid FC
                : toto (b, a + a, b) ? (qux (b, a))  // valid FC
                :                       a;           // valid LX
    }

    int baz (int a, int b) {
      // this expression is not a valid CX,
      // because the second operand is not a valid FC
      return goto foo (a + b, a) ? toto (a + a, b, a)  // mismatched function type
                : foo (b, a + a) ? (qux (b, a))        // (valid FC)
                :                  ((0));              // (valid LX)
    }
```

The conditional expression is a valid instance of *CX* only if both its second and third operands recursively consist of valid *operand values*.

EXAMPLE 6 In general, Standard Library functions cannot portably be used as the target of a `return goto` statement because they could be implemented either as a function call or as a macro expanding to an expression with some other structure:

```
#include <ctype.h>

int my_isalnum (int c) {
  if (condition) {
    return goto isalnum (c);  // potentially invalid if isalnum is provided
  }                           // as a macro in ctype.h, which is possible
  else return some_fallback;
}

extern long long llabs (long long);  // non-header declaration

long long my_llabs (long long x) {
  if (condition) {
    return goto (llabs) (x);  // suppress any possible expansion and
  }                           // use the external declaration
  else return some_fallback;
}
```

Standard library functions may have other properties making them ineligible for tail calls for implementation-defined reasons.

Add a new entry to the end of the list in paragraph 1 of 6.10.10.4 "Conditional feature macros":

`__STDC_TAIL_CALLS__` The integer constant 1, intended to indicate that there is some set of functions for which the implementation supports generating direct tail calls using the `return goto` statement; or the integer constant 0, intended to explicitly indicate that there are no such calls supported by the implementation.

No modifications are made to the Standard Library.

6 Extended Constant Expressions

6.1 Introduction

C requires that objects with static storage duration are only initialized with constant expressions. The rules for which kinds of expression may appear as constant expressions are quite restrictive and mostly limit users to using macro names for abstraction of values or operations. Users are also limited to testing their assertions about value behavior at runtime because `static_assert` is similarly limited in the kinds of expressions it can evaluate at compile-time.

This Specification adds a new function specifier to C, `constexpr`, as introduced to C++ in C++11, and introduced to C in C23 as a storage-class specifier for objects. This specifier is here added to functions separately from its role as a storage class for objects, and intentionally keeps the functionality minimal to avoid undue burden on lightweight implementations.

This feature was first discussed during the January 2022 meeting of WG14, and the meeting established some starting directions for the feature set:

— WG14 would like the `constexpr` specifier to be added to C23, for objects only;
— WG14 would like UB to be prohibited from constant expressions in general;
— there was not consensus to add any additional operators (element access) to the set that can be used for constant expressions, for C23;
— there was not consensus to add the `constexpr` specifier for functions, to C23;
— there was strong consensus to add the full `constexpr` feature for objects, extended operators (element access), and function definitions, in some future version of C after C23.

At the July 2022 meeting of WG14, the `constexpr` keyword was adopted as a storage-class specifier for objects, which introduced the feature to C23. The specifier as it appears in the final published version of C23 does not allow for subscript access to the values in array objects, but does allow for members (and whole values) of structure and union objects to be used in constant expressions.

The rationale for, and impact of, the inclusion of *constexpr functions* as well as *constexpr objects* was explained and discussed in WG14 document N2917 (“The `constexpr` specifier, v2”).

6.2 Overview of extended constant expressions and definitions

For the purpose of this Specification, a *constexpr object* is any object defined with the `constexpr` specifier as part of its declaration specifiers. The value of a constexpr object may be accessed as part of any kind of C constant expression (which does not include preprocessor expressions), assuming it has appropriate type. Constexpr objects are a standardized feature of C23 and are described in ISO/IEC 9899:2024.

For the purpose of this Specification, a *constexpr function* is any function defined with the `constexpr` function specifier. The keyword for the `constexpr` storage class is reused, and given a second role as a function specifier. A constexpr function definition is subject to stricter constraints than other function definitions, but is *not* subject to the constraint that forbids them from being called from within constant expressions.

When a constexpr function appears within a constant expression, it is evaluated by *invocation substitution*, described below. The result expression of the function is inlined into the calling context, avoiding the need to treat parameters and local variables separately from named constants. This has useful properties for simplifying the evaluator.

For the purpose of this Specification, a constant expression is one of: a named constant; an integer constant expression; an arithmetic constant expression; an address constant expression; a structure or union constant expression; an array constant expression; a null pointer constant; or any other form of constant expression accepted by the implementation. Constant expressions are described in Section 6.6 “Constant expressions” of C23.

A constant expression does not contain any diagnosable undefined behavior. An expression containing undefined behavior in an evaluated branch is not a constant expression for any purpose.

This Specification therefore generalizes C constant expressions of all kinds, to also include element access to array objects defined with the `constexpr` storage class, and to include calls to functions defined with the `constexpr` function specifier.

6.2.1 `constexpr` array element access

C23 allows objects of any type to be constexpr objects, but does not change the expression rules to necessarily make these useful outside of initialization contexts (which can use the `[]` and `->` operators in combination with & to create an address constant, but not to read a value). This specification adds the ability to make use of the element values of array objects, which can be declared in C23 (and their values can even be copied as a group by initialization if they are members of a containing structure constant), but cannot be accessed within constant expressions.

For the purpose of this specification, all kinds of constant expression may also make use of the subscript operator `[]`. Constant expressions may not make use of the `*` or `->` operators to access the value of an object, because the identity and therefore storage class of the object designated by such an expression is not necessarily traceable.

The `[]` operator may be used with the added restriction that the "array" operand (the pointer-typed operand) must be the name of a constexpr object declared with complete array type. This is the sole exception to the rule that otherwise prohibits the use of `*` to access the value of an object (`[]` is otherwise defined in terms of `*`). The index operand must be an integer constant expression, and its value must be greater than or equal to zero, and less than the number of elements of the array. One restriction that can be removed is the requirement for the array not to have `register` storage class so long as the usage is unevaluated or constant.

Therefore, the `[]` operator must have either a (possibly-parenthesized) identifier as the leading postfix-expression, or an element access expression itself designating an array element of either a structure object or a containing array, or a generic selection selecting one of the previous forms, rather than a generalized address constant expression of pointer type.

This is symmetrical with the relaxation on the use of the function call operator with a function name in the postfix position.

6.2.2 `constexpr` functions

A function declared with the `constexpr` function specifier is subject to stricter restrictions than other C functions, taken from the quite restrictive set of rules used by C++11 (ignoring those rules that are not applicable to C, and with some minor usability enhancements).

The function body may only contain:

— null statements (plain semicolons)
— `static_assert` declarations
— `typedef` declarations
— tag declarations (not in C++11)
— object definitions with the `constexpr` storage class (not in C++11)
— initialized definitions of objects with automatic storage duration (not in C++11)

...in addition to exactly one return statement, which is not allowed to treat any lvalue as *modifiable*. The function must return a non-`void` value. The function may invoke itself recursively in a conditionally-evaluated expression branch. (Note that if this recursive invocation is *not* in a conditional branch, the function will definitely never halt!)

The return statement may be a `return goto` statement if supported by the implementation. (`return goto`'s support for the *CX* and *LX* expression classes is intended to enable this usage.)

A constexpr function is implicitly also an inline function, allowing it to be defined in a header. All other considerations for the use of the C `inline` specifier apply in the same way to C constexpr functions (C++ has slightly different inline rules, which *do not* apply).

An invocation of a constexpr function with arguments that are all themselves constant expressions is a constant expression. A constexpr function may also be called with non-constant arguments, and in that case behaves like any other function call; such a call is not a constant expression. The address of a constexpr function may be taken and used as any other function pointer can; this does not preserve the `constexpr` specifier, which is not part of the function's type (in the same way that `inline` is not part of the type).

All kinds of constant expression may therefore make use of the function call operator in addition to the other operators permitted by C23 in a constant context, with the added restriction that the call must be to the name of a defined constexpr function. Generalized function pointers cannot be invoked in a constant expression because the `constexpr` specifier is not preserved as part of a called function's type. Therefore, the function call operator must have a (possibly-parenthesized, possibly generic-selected) identifier as the leading postfix-expression, rather than a generalized address constant expression of pointer-to-function type.

This is symmetrical with the relaxation on the use of the `[]` operator with an array name in the postfix position. Because of invocation substitution, both arrays and functions passed by name as arguments to a constexpr function can be subscripted or called respectively within the function, so long as they are array constants or constexpr functions in the calling context.

Further discussion of the rationale for adopting the C++11 ruleset (as far as relevant to C) can be found in WG14 document N2917 ("The `constexpr` specifier, v2"). Of note is that this ruleset **does not require any interpreter semantics** to be added to the C translator, as all value-uses are side-effect-free and all function calls can therefore be expanded essentially like macros, by direct, scope-aware, substitution of the result expression into the call site, and then evaluated using the existing constant evaluation engine required for C23.

An implementation choosing to provide more fully-featured constexpr functions, as found in C++14 or later (mutable local state, loops, etc.), is therefore very strongly encouraged to also provide the user with portability warnings. See Clause 7 for the specification of such a feature set that builds directly upon the expression evaluator and aims to avoid such problems.

6.2.3 Local variable declaration

The most notable departure from the C++11 ruleset introduced by this clause is the ability to use local variables (initialized objects declared at block scope with automatic storage duration).

The rationale for this is that a parameter is *also* a local variable, and therefore any function making use of a local can be *trivially* rewritten to a function invoking a helper with an extra parameter:

```
constexpr int foo1 (int x, int y) {
    int z = x + 1, w = x * y;
    int u = y * z, v = u + 1;  // u is complete at the comma

    return (u + v + w) - (x + y);
}
```

is exactly equivalent to

```
constexpr int foo2 (int x, int y) {
    // we can get away with rewriting two declarations into one call
    // here only because they do not depend on each other
    return foo2_helper1 (x, y
                       , /*z*/ x + 1, /*w*/ x * y);
}

constexpr int foo2_helper1 (int x, int y  // (assume forward declarations were inserted)
                          , int z, int w) {
    // more generally, each declaration needs a new function
    return foo2_helper2 (x, y, z, w     // because subsequent objects may
                       , /*u*/ y * z);  // depend on it, like v does on u
}
constexpr int foo2_helper2 (int x, int y
                          , int z, int w
                          , int u) {
    return foo2_helper3 (x, y, z, w, u
                       , /*v*/ u + 1);
}
constexpr int foo2_helper3 (int x, int y
                          , int z, int w
                          , int u
                          , int v) {
    // finally everything is named and in scope
    return (u + v + w) - (x + y);
}
```

Or using function literal notation, for brevity and readability:

```
constexpr int foo3 (int x, int y) {
    return [] (int z
             , int x, int y) {
        return [] (int w
                 , int x, int y, int z) {
            return [] (int u
                     , int w, int x, int y, int z) {
                return [] (int v
                         , int u, int w, int x, int y, int z) {
                    return (u + v + w) - (x + y);
                } (u + 1
                 , u, w, x, y, z);
            } (y * z
             , w, x, y, z);
        } (x * y
         , x, y, z);
    } (x + 1
     , x, y);
}
```

This only holds if all block scope objects within a constexpr function are considered *underspecified.* This prevents them from relying on any indeterminate values and requires every object to be initialized, and allows them to introduce a substitutable *binding* for their initializing expression in the same way as a parameter.

Invocation substitution does not require implementations to actually rewrite functions to use helpers, as the definition specifies substitution in two steps.

6.3 Detailed changes to ISO/IEC 9899:2024

The modifications are ordered according to the clauses of ISO/IEC 9899:2024 to which they refer. If a clause of ISO/IEC 9899:2024 is not mentioned, no changes to that clause are needed. New clauses are indicated with (NEW CLAUSE), however resulting changes in the existing numbering are not indicated; the clause number *mm.nn***a** of a new clause indicates that this clause follows immediately clause *mm.nn* at the same level. Bolded text within an existing clause is new text.

Add two new bulleted entries to 5.3.5.2 "Translation limits":

> — 64 nested constexpr function invocations
> — 8192 nested constant expressions within the evaluation of a single constant expression

Note that C++20 specifies larger values of 512 and 1048576 respectively. The C values are arbitrary but aim to respect the lower minimum translation requirements expected for a C implementation.

Modify 6.3.3.1 "Lvalues, arrays, and function designators":

Modify the last sentence of paragraph 3:

> **Except when the expression is evaluated as part of a constant expression or appears within an unevaluated expression context,** if the array object has register storage class, the behavior is undefined.

This allows the use of `register`-class array elements within unevaluated expressions and within constant expressions.

Modify 6.6 "Constant expressions":

Paragraph 3, relax the constraint against function calls:

> Constant expressions shall not contain assignment, increment, decrement, or comma operators, except when they are contained within a subexpression that is not evaluated. **If a function-call operator appears in an evaluated part of a constant expression, the postfix-expression designating the function to call shall consist only of the (possibly-parenthesized) identifier of a function declared with the `constexpr` function specifier, or a generic selection evaluating to such, and all argument expressions shall themselves be constant expressions. The function to call shall be defined in the translation unit before it is evaluated, directly or indirectly, from any context requiring a constant expression.**

Modify paragraph 6 to include arrays:

> A compound literal with storage-class specifier `constexpr` is a *compound literal constant*, as is a postfix expression that applies the `.` member access operator to a compound literal constant of structure or union type, **and a postfix expression that applies the `[]` subscript operator to a compound literal constant of array type,** even recursively. A compound literal constant is a constant expression with the type and value of the unnamed object.

Modify paragraph 7 to include arrays in the same way:

> ... is a *named constant*, as is a postfix expression that applies the . member access operator to a named constant of structure or union type, **and a postfix expression that applies the `[]` subscript operator to a named constant of array type,** even recursively. **The result of a parenthesized expression or generic selection, is a named constant if all operands to the expression are named constants.**

Paragraph 8, include function calls returning integer values in the first sentence:

An *integer constant expression*[116)] shall have integer type and shall only have operands that are integer constants, named and compound literal constants of integer type, character constants, **calls to constexpr functions with integer return type and constant arguments,** `sizeof` expressions whose results are integer constants, `alignof` expressions, and floating, named, or compound literal constants of arithmetic type that are the immediate operands of casts.

Paragraph 10, include function calls in the first sentence:

An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, named or compound literal constants of arithmetic type, character constants, `sizeof` expressions whose results are integer constants, `alignof` expressions, **and calls to constexpr functions that return a value with arithmetic type**.

Move paragraph 13 to follow existing paragraph 11, and add references to constexpr functions and array constants:

A *structure or union constant* is a named constant or compound literal constant with structure or union type, respectively; **or a call to a constexpr function that returns a value with structure or union type. An *array constant* is a named constant or compound literal constant with array type, or a call to a constexpr function whose return expression designates an array constant.**[footnote)]

Modify existing paragraph 12 (and footnote 118) to allow element access by subscript:

... but the value of an object shall not be accessed by use of **the * or -> operators.**[118)] **A value shall only be accessed by use of the array-subscript operator `[]` if its pointer-typed operand is an array constant (implicitly converted to a value of pointer type as specified by 6.3.2.1), and its integer-typed operand is an integer constant expression with a value greater than or equal to zero and less than the number of elements of the array type.**[footnote2)]

118) **therefore a constexpr function may receive an *address constant* as an argument, but may not dereference it using *.** Named constants or compound literal constants with ...

footnote2) an address constant that is not the address of an array defined with the `constexpr` specifier can be passed to and returned from a constexpr function, but not subscripted with `[]`.

Add a new paragraph after paragraph 12:

An address constant that is returned from a constexpr function remains an address constant in the calling context.

Modify existing paragraph 15:

Starting from a structure or union constant, the member-access `.` operator may be used to form a named constant or compound literal constant as described previously in this subclause. **Starting from an array constant, the array-subscript operator `[]` may be used to form a named constant or compound literal constant as described previously in this subclause.**

Add a new final paragraph after paragraph 17:

If the evaluation of any subexpression invokes undefined behavior, the entire containing expression is not a constant expression.

Add a forward reference:

> array declarators (6.7.7.3), **function specifiers (6.7.5),** initialization (6.7.11).

Modify 6.7.1 “Declarations”, paragraph 12:

> A declaration such that the declaration specifiers contain no type specifier, or that is declared with `constexpr`, **or declares an object within a constexpr function,** is said to be *underspecified.* If such a declaration is not a definition, if it declares no or more than one ordinary identifier **with no type specifier or with `constexpr`**, if the declared identifier already has a declaration in the same scope, if the declared entity is not an object, or if anywhere within the sequence of tokens making up the declaration identifiers that are not ordinary are declared, the behavior is implementation-defined.[123)]

Add a forward reference:

> ... type qualifiers (6.7.4)**, function specifiers (6.7.5)**.

Modify 6.7.2 “Storage-class specifiers”:

Add one new paragraph after paragraph 5:

> The `constexpr` specifier is treated as a function specifier when applied to a function declaration.

Add a forward reference:

> ... type definitions (6.7.9)**, function specifiers (6.7.5)**.

Modify 6.7.2 “Storage-class specifiers”, the last sentence of footnote 127:

> ... Thus, the only operator that can be applied to an array declared with storage-class specifier `register`, **except when forming a *named constant*,** is `sizeof` and the typeof operators.

Modify 6.7.5 “Function specifiers”:

Paragraph 1, add the `constexpr` specifier:

> *function-specifier*:
> `inline`
> `_Noreturn`
> **`constexpr`**

Add a new subheading and paragraphs after paragraph 1:

> **Description**
>
> A function declared with an `inline` function specifier is an *inline function.* A function declared with a `constexpr` function specifier is a *constexpr function.*
>
> A constexpr function is implicitly also an inline function.

Add five new paragraphs after paragraph 4:

> A constexpr function shall not have `void` return type.

A constexpr function shall return a value, and shall not contain any statement except a single return statement with an expression, and zero or more:

- null statements
- static assertions
- typedef and tag declarations
- definitions of objects with the `constexpr` storage class
- definitions of objects with automatic storage duration

A constexpr function shall not contain any specification of a variably-modified type, apart from the declaration of parameters declared with an array type before adjustment,footnote1) except within the operand of `sizeof` or `alignof`.footnote2)

footnote1) this array type is converted to an undimensioned pointer type by adjustment as described in 6.7.7.4, and is supported even when `__STDC_NO_VLA__` is defined equal to 1; no VLA or variably-modified type is defined by such a parameter.

footnote2) therefore if a variably-modified type appears in the operand of a typeof operator, this typeof operator will ultimately appear within the operand of `sizeof` or `alignof`.

The function definition shall not contain any subexpressions that would cause invocation substitution (defined subsequently in this subclause) to produce an expression which is not a constant expression.footnote)

footnote) even if it is never actually called with constant arguments.

If any declaration of a function has a `constexpr` specifier, then all of its declarations shall contain a `constexpr` specifier.

Delete the first sentence of paragraph 6 (which has been moved to the top of the section).

Add seven new paragraphs after paragraph 8:

A call to a constexpr function identifier with arguments that are all constant expressions can be used in contexts such as static assertions or initialization of objects with static storage duration, after it has been defined.

When invoked with constant arguments from a context that requires a constant expression, the result of a constexpr function is a constant expression with a category as follows:

- the result of a constexpr function that has an integer return type is an integer constant expression;
- the result of a constexpr function that has non-integer arithmetic type is an arithmetic constant expression;
- the result of a constexpr function that has a pointer or `nullptr_t` return type is an address constant;
- the result of a constexpr function that has a structure or union return type is a structure or union constant, respectively;
- the result of a constexpr function whose return expression designates an array constant, and that has a return type of pointer to the designated array constant's element type, is an array constant in the calling context, despite having pointer type.

An implementation may allow constexpr functions to return any additional categories of constant expression that it defines (6.6).

A call to a constexpr function produces the same result as a call to an equivalent non-constexpr function in all respects except that a call to a constexpr function can appear in a constant expression.footnote)

footnote) therefore, when a constexpr function is invoked from a context that does not require a constant expression, the result of the evaluation is the same as if it had been defined without the `constexpr` specifier, particularly with respect to the floating-point environment.

A constexpr function does not modify any state, or observe any state outside of its own argument values. Any constexpr function implicitly fulfills the requirements of the `[[unsequenced]]` or `[[reproducible]]` attributes.

A constexpr function invoked during evaluation of a constant expression[footnote1)] is evaluated by *invocation substitution*:

- for each parameter, the corresponding argument expression is evaluated,[footnote2)] implicitly converted to the parameter type, and substituted for each use of the parameter object within the function body;[footnote3)] then
- for each use of an object with automatic storage duration declared inside the function, the identifier expression is replaced by a compound literal constant of the same type and with the `constexpr` storage class, initialized by the initializer of the substituted object; then
- the `return` statement's expression is converted to the return type of the function, and substituted for the function call expression in the containing constant expression, where it is evaluated according to the rules of 6.6.[footnote4)]

footnote1) a call to a constexpr function that appears within an unevaluated subexpression is not evaluated and does not invoke undefined behavior.

footnote2) therefore, if an argument expression invokes undefined behavior, the call expression invokes undefined behavior, even if the corresponding parameter is only substituted into unevaluated subexpressions within the function body; the call expression is consequently not a constant expression.

footnote3) a parameter with pointer type may be used with the subscript operator if its argument value is an array constant, because implicit conversion of an array does not remove the array constant property.

footnote4) this recursively implies that any potentially-evaluated function calls within the body of the function are also to constexpr functions, because they are being evaluated in a constant context.

The lexical scope of identifiers is not affected by invocation substitution, and any identifiers of entities (such as tags or constants) defined within the scope of the constexpr function and referenced in the fully-substituted return expression continue to refer to those same entities.

Add two NOTEs:

NOTE: a constexpr function may also be called with non-constant values or have its address taken, in which case it behaves like any other function. The type of the function is not affected by the `constexpr` specifier.

NOTE: the `constexpr` specifier may be applied to a forward declaration, but a call to it is not a constant expression unless the definition is visible when the outermost constant-expression containing the (possibly-nested) call is evaluated.

Add two Recommended Practice paragraphs after existing paragraph 9:

Implementations are encouraged to issue a diagnostic message when a constexpr function is called with constant arguments and the definition is not visible, as this is never a constant expression.

Implementations are encouraged to issue a diagnostic if the declared size of an array parameter before adjustment does not match the actual size of the array constant passed as the argument.

Add four examples:

EXAMPLE 3 A constexpr function can be called as part of a constant expression from a call site appearing syntactically before its definition, so long as the originating call appears after the definition:

```
constexpr int inner (int x);

constexpr int outer (int x) {
  return inner (x + 1);         // syntactic use before definition
}
static_assert (outer (3) == 6); // not a constant expression
                                // because inner is not yet defined

constexpr int inner (int x) {
  return x + 2;
}
```

```
static_assert (outer (3) == 6); // valid constant expression, because
                                // inner is defined before use by outer
```

EXAMPLE 4 Invocation substitution does not affect the lexical scope of any identifiers that are not substituted:

```
constexpr int has_scoped (int x) {
  typedef int Int;
  static constexpr int K = 10;
  return K + (sizeof (Int) * x);
}

// valid because Int and K are still looked up
// inside the scope of has_scoped:
static_assert (has_scoped (7) == (10 + (sizeof (int) * 7)));
```

EXAMPLE 5 Because invocation substitution replaces the parameter variable with its argument expression, if the argument expression was a structure, union or array constant, or the identifier of a constexpr function, the parameter will be evaluated as the same constant category in the calling context:

```
constexpr int array[] = { 1, 2, 3, 4 };
constexpr int func (int x, int y) { return x + y; }
constexpr struct S { int u, v; } s1 = { 5, 6 };

constexpr int use (int * p,
                   typeof (int (int, int)) * f,
                   struct S s2) {
  return p[2]
       + f (s2.u, s2.v);
}

static_assert (use (array, func, s1)  // valid
            == (3 + 5 + 6));
// because it expands to the equivalent of:
static_assert ((array)[2] + (func) (s1.u, s1.v)
            == (3 + 5 + 6));
```

EXAMPLE 6 Local variables are substituted by an identically-initialized compound literal:

```
constexpr int locals1 (int x, int y) {
  int z = x + 2;
  return z * y;
}
static_assert (locals1 (4, 6)
            == ( (constexpr int){ (4) + 2 } * (6) ));

constexpr int locals2 (int x, int y, int z, int w) {
  int arr[] = { x + y, z * 2, w + 1 };
  return arr[0] + arr[1] + arr[2];
}
static_assert (locals2 (1, 2, 3, 4)
            == ((1 + 2) + (3 * 2) + (4 + 1)));
// substitutes to the equivalent of:
static_assert ( (constexpr int[]){ 1 + 2, 3 * 2, 4 + 1 }[0]
              + (constexpr int[]){ 1 + 2, 3 * 2, 4 + 1 }[1]
              + (constexpr int[]){ 1 + 2, 3 * 2, 4 + 1 }[2]
            == 14);
```

Add a forward reference:

Forward references: function definitions (6.9.2)**, attributes (6.7.13)**.

Add a definition to 6.10.10.4 “Conditional feature macros”:

`__STDC_CONSTEXPR_TS__` The integer constant 1, intended to indicate conformance to the specifications for constexpr functions in TS 25007.

No modifications are made to the Standard Library.

7 Extended constexpr functions

7.1 Introduction

After C++11 introduced heavily-restricted constexpr functions similar to those specified by Clause 6, users immediately started asking for more flexibility in what could be expressed. Particularly users without a functional programming background disliked the restriction to recursion and ternary expressions for implementing anything with any kind of looping behavior.

C++14 therefore relaxed the restrictions on constexpr functions to allow for more kinds of local statefulness, making statements other than `return` and declarations usable within the definition. Such functions still have no external side effects, but being able to assign to local variables does simplify expressing loops and related constructs.

This request was *immediately* made by members of the C Community as well, even before the first draft of TS 25007 was published. Although the original intent was not to explore this path for the Standard, the overwhelming response from the Community means that it cannot be ignored either. Therefore, additional extensions to constexpr function definitions are specified in a second separate clause so that the Community has greater flexibility in what it chooses to adopt. This increases the utility of the TS as a tool for exploring user preferences and gathering implementation experience.

The original reason for staying strictly within the C++11 rule set was to ensure that constexpr functions could always be evaluated by simple inline substitution. With no assignment expressions and no statements, the return expression can be inlined directly into a caller context by invocation substitution, whereby parameters have their values simply copied into the inlined expression. This strategy means that **no additional interpreter machinery** (except for 5-10 lines to inline the expression) is needed in order to support basic constexpr functions beyond the existing evaluator already demanded by C23 constant expression evaluation (and the versions of the language before it).

There is great concern in the Committee that the Standard should *never* require implementers to have to ship a stateful interpreter VM as part of a compiler offering, as this is a high burden on lightweight implementations.

In order to support this stance, the following extended constexpr function functionality is defined entirely in terms of rewrites to the expression-oriented forms permitted by Clause 6. Every statement provided by Clause 7 can be rewritten and inlined into a single-`return` function form, in some cases extracting intermediate steps into helper constexpr function bodies.

This enables an implementation to provide statement support without providing an interpreter. This also provides a better definition of the semantics of each statement by ensuring every form is expressed in terms of expression evaluation.

An implementation that does wish to support local statements inside constexpr functions is in no way required to implement the functionality in this way, so long as the end result has identical semantics: if adding a statement-oriented interpreter is an easier implementation strategy, that is a decision for the tool maintainer. (Actually evaluating the rewritten forms as-specified is likely to be extremely inefficient.)

7.2 Informative semantic description

This clause extends the definition of constexpr functions to allow the following statement kinds to appear within the body of such a function, in addition to constant/type declarations and the single return statement allowed by Clause 6:

— compound statement, null statement
— local variable declaration with initialization
— expression statement, where the top-level operator is an assignment or increment operator
— `if` statement
— `for`, `while` and `do` iteration statements
— `break` and `continue` statements
— `goto`
— `switch` statement

The semantics of each of these is defined in terms of rewrites to simpler statement forms, eventually reaching a fully-rewritten form consisting entirely of calls to function definitions whose bodies satisfy the basic constraints expressed in Clause 6, of a single return statement accompanied by translation-time declarations only. The order of clauses aims to present more complex rewrites later in the sequence.

In order to simplify the description of the rewrites, the “functional update” feature from Clause 4.2 and the “function literal” feature from Clause 4.3 are used as though they were built-in language features. These can in turn be rewritten and eliminated according to the descriptions provided in Clause 4.

This is a *continuation*-oriented approach. For all statements other than jump statements, the *continuation* of the current statement is the lexically-subsequent *block-item* of the containing statement’s *block-item-list*; or if it is the last item in a *block-item-list*, the *continuation* is the same as the *continuation* of the enclosing statement.

The static declarations already permitted by Clause 6 do not form the *continuation* of any statement (as they do not form part of regular control flow), and are ignored for the purposes of identifying the next *block-item*, as are labels.

A return statement has no local continuation and ends execution of the containing function. Every other declaration or statement shall have a non-empty continuation. A function definition’s outermost compound statement has no continuation and therefore ends with a return statement.

Rewriting begins with the outermost compound statement forming the body of the function definition. Each statement kind defines the order in which its subsequent and nested statements are rewritten. The end result shall be a single return statement. During rewriting, because the order proceeds in reverse, the continuation is always already a return statement.

The scope of identifiers introduced by declarations as used in expressions is not affected by rewriting unless otherwise specified (i.e. if a typedef, constant, or possibly-predefined identifier is used in an expression, it will continue to be valid in that expression with its original meaning, regardless of how rewriting moves the expression around).

7.2.1 Compound statements, null statements, and labels

Feature: compound and null statements provide structure for imperative programming constructs and to create new explicit lexical scopes. Labels name a statement for use as a jump target.

Description: The null statement is discarded.

A label is not a statement and does not have a continuation of its own. Labels may be used in the same ways as in the core language.

After rewriting has converted any given statement to a `return` statement, any label naming the original statement names the rewritten `return` statement. As per 7.1 this does not affect the scope or visibility of the label (although it may exist in a different generated function, it is still the same jump target).

The compound statement is rewritten by rewriting the items in its *block-item-list* in reverse lexical order (starting with the last *block-item*), so that the continuation of each item has already been rewritten when it is being handled.

After this process is complete, there will be at most one *block-item* remaining, which replaces the compound statement within the containing statement, or forms the return statement of a function definition's body.

As in the core language, a compound statement introduces a new explicit lexical scope.

7.2.2 Local variable declaration

Feature: objects with block scope and automatic storage duration may be declared in a constexpr function definition. (This feature is also supported by Clause 6, using the simpler substitution mechanism.)

Description: all block scope objects within a constexpr function are considered *underspecified.* This prevents them from relying on any indeterminate values and requires every object to be initialized. The implementation is encouraged to define declarations that declare more than one identifier as supported where they are not already *underspecified* according to C23.

A local variable adds a new identifier binding for a value into the subsequent scope. Each declaration marks the beginning of a new scope, regardless of block depth (this is the same as in Standard C - scopes extend "downward" lexically). Rewriting can therefore always insert a new implicit compound level around the start of a declaration continuing until the end of the block scope in which the declaration appears syntactically, which forms the core mechanism.

Since a parameter also introduces an identifier binding, any declaration with automatic storage duration can be rewritten to be expressed in terms of parameters to the continuation, extracted into a new function body:

```
constexpr int foo1 (int x, int y) {
    int z = x + 1, w = x * y;
    int u = y * z, v = u + 1;  // u is complete at the comma

    return (u + v + w) - (x + y);
}

// is exactly equivalent to

constexpr int foo2 (int x, int y) {
    // we can get away with rewriting two declarations into one call
    // here only because they do not depend on each other
    return foo2_helper1 (x, y
                       , /*z*/ x + 1, /*w*/ x * y);
}
```

```
constexpr int foo2_helper1 (int x, int y  // (assume forward declarations were inserted)
                           , int z, int w) {
    // more generally, each declaration needs a new function
    return foo2_helper2 (x, y, z, w     // because subsequent objects may
                        , /*u*/ y * z);  // depend on it, like v does on u
}
constexpr int foo2_helper2 (int x, int y
                           , int z, int w
                           , int u) {
    return foo2_helper3 (x, y, z, w, u
                        , /*v*/ u + 1);
}
constexpr int foo2_helper3 (int x, int y
                           , int z, int w
                           , int u
                           , int v) {
    // finally everything is named and in scope
    return (u + v + w) - (x + y);
}
```

Or, henceforth using function literal notation, for brevity and readability:

```
constexpr int foo3 (int x, int y) {
    return [] (int z
             , int x, int y) {
        return [] (int w
                 , int x, int y, int z) {
            return [] (int u
                     , int w, int x, int y, int z) {
                return [] (int v
                         , int u, int w, int x, int y, int z) {
                    return (u + v + w) - (x + y);
                } (u + 1
                  , u, w, x, y, z);
            } (y * z
              , w, x, y, z);
        } (x * y
          , x, y, z);
    } (x + 1
      , x, y);
}
```

Examples of rewrites given below will generally leave the "captures" list of extended parameters implicit, or compressed (e.g. `Locals ...`) for the sake of the example. It should be assumed that any local variables used in a block or statement that has been extracted to a helper lambda are passed as part of the extended argument list, as shown in full above.

A local declaration therefore rewrites as if by extracting its continuation to a new function body, which is parameterized with the new object's identifier and called with the initializing value.

A declaration that is the last *block-item* in the *block-item-list* is discarded and replaced by its continuation.

A declaration that defines an array is rewritten as-if it defines an object of an appropriate "box" type, and all subsequent expressions making use of the declared identifier are rewritten to access the "box" member:

```
int arr[] = { 1, 2, 3 };
return arr[1] + arr[2];

// rewrites as
```

```
struct ArrBox { int box[3]; };
return [] (ArrBox arr) {
    return arr.box[1] + arr.box[2];
} ((struct ArrBox){ .box = { 1, 2, 3 } });

// and NOT as

return [] (int arr[3]) { // <-- this is a pointer after adjustment
    return arr[1] + arr[2];
} ((int[3]){ 1, 2, 3 }); // <-- this is implicitly converted
```

Therefore rewriting does not introduce new instances of array-to-pointer adjustment.

7.2.3 Modifying assignment

Feature: a modifiable object or sub-object defined with automatic storage duration may be updated by means of an assignment, increment, or decrement operator.

Description: a statement may be an expression statement where the top-level operator, ignoring any parenthesization and after generic selection, is an assignment, increment, or decrement operator. These operators may not appear in subexpression operands nested within the expression, except in the unevaluated operand of `sizeof`, `alignof`, or the typeof operators and any discarded generic associations.

Because there is no nested appearance, no semantic distinction is made between prefix and postfix increment or decrement operators.

The increment operator is treated as though it is an assignment operator expressing `E += 1`. The decrement operator is treated as though it is an assignment operator expressing `E -= 1`.

A compound assignment is then treated as though it was written as a simple assignment:

```
L @= E
// unconditionally becomes
L = L @ E
```

NOTE Because there are no internal or external side effects allowed in `L`, there is no observable difference in repeating the evaluation.

The lvalue expression `L` shall designate an object defined in the containing function with automatic storage duration, or a sub-object of such an object accessed by means of the `.` or `[]` operators. The lvalue expression `L` shall not designate an element of an array passed as a pointer parameter to the containing function (which is not an object defined locally), or any sub-object of such an element.

When the lvalue expression `L`, ignoring parenthesization and after generic selection, is a *postfix-expression* designating a member or element of an object, it is rewritten to use the equivalent functional update form, by removing all postfix-expressions from the lvalue and using them directly as the sub-object designator:

```
obj.a1[expr].m2 = X;

// is rewritten as

obj = (typeof (obj)){ obj; .a1[expr].m2 = X };
```

The expression is now in a form `ID = expr` where `ID` is the identifier designating a local object. This is rewritten into a declaration of a new identifier of the same type, and all references to the identifier `ID` in the continuation are replaced by references to the new identifier:

```
L = expr;
... (L + 1, ...

// is rewritten as

typeof (L) L@1 = expr;
... (L@1 + 1, ...
```

The statement is then finally re-examined for rewriting as a declaration.

NOTE whole-object copying does not change the semantics of subsequent expressions as the object does not have its address taken with &.

7.2.4 `if` statement

Feature: two statements may be selected between by a controlling expression, evaluating only one sequence of sub-statements.

Description: any `if` statement is informally equivalent to a conditional expression that selects the result of two function calls, each encapsulating the statement for the respective branch:

```
if (x > 10)
    STMT1;
else
    STMT2;
CONT;

// is exactly equivalent to

x > 10 ? [] (int x) { STMT1; } (x)
       : [] (int x) { STMT1; } (x);
CONT;
```

This forms the basis of the rewrite strategy.

An `if` statement is therefore rewritten by moving a copy of its continuation to the end of each of its *secondary-blocks*:

```
if (COND)
  STMT1;
else
  STMT2;
CONT;

// is rewritten as

if (COND) {
    STMT1;
    CONT;
} else {
    STMT2;
    CONT;
}
```

Both *secondary-blocks* are then rewritten with their new continuation, and each encapsulated in a function body parameterized with the values of the objects in scope (as-if to redeclare them using the rules from 7.1.1), which are finally used as the second and third operands of a conditional expression forming the operand expression to `return`:

```
// and therefore as

return COND ? [] (Local l) {
```

```
               // (after local rewriting to thread STMT1 into CONT)
               return [STMT1 with CONT];
              } (l)
            : [] (Local l) {
               return [STMT2 with CONT];
              } (l);
```

An `if` statement with only one branch creates an implicit empty *secondary-block* for the `else` branch before appending to it:

```
if (COND)
  STMT1;
CONT;

// is rewritten as

if (COND) {
    STMT1;
    CONT;
} else {
    CONT;
}

// and therefore as

return COND ? [] (Local l) {
               return [STMT1 with CONT];
              } (l)
            : [] (Local l) {
               return CONT;
              } (l);
```

NOTE an `if` statement rewrites to a `return` because its continuation already encapsulates evaluation of the remainder of the function body.

NOTE discarding one branch when the condition is a constant is a valid optimization, but has no effect on the evaluation semantics.

7.2.5 Iteration statements

Feature: a statement may be repeated zero or more times for its local side effects before continuing execution of the containing block.

Description: any loop can be rewritten as a recursive function that selects between returning immediately or re-entering itself.

A `while` statement is rewritten as a function encapsulating an `if` statement that selects between the `while` statement's continuation, and the loop body statement with a recursive call to the generated function added as its *generated local continuation*:

```
STMT1;
while (x < 10)
  STMT2;
CONT;

// is rewritten as

STMT1; // executed once, not part of the recursion

return [] (int x) {        // <-- recursion jumps to here
    if (!(x < 10)) {       // not-branch first
        [return] CONT;     // (CONT is already a return)
    } else {
        STMT2;
        return _Recur (x); // local continuation of body
    }
} (x);
```

NOTE The rewrite rules for `if` will implicitly place all subsequent statements into the exit branch via the continuation.

The body of the added encapsulating function is then re-examined for rewriting.

A `do` statement is rewritten as a function encapsulating the loop body, followed by an `if` statement that selects between the `do` statement's continuation, and a recursive call to the generated function as the *generated local continuation*:

```
STMT1;
do
  STMT2;
while (x < 10);
CONT;

// is rewritten as

STMT1; // executed once, not part of the recursion

return [] (int x) {        // <-- recursion jumps to here
    STMT2;

    if (!(x < 10)) {       // not-branch first
        [return] CONT;     // (CONT is already a return)
    } else {
        return _Recur (x); // repeat loop
    }
} (x);
```

The body of the added encapsulating function is then re-examined for rewriting.

A `for` statement is rewritten as a `while` loop, extracting *for-clause-1* to a new *block-item* at the same level immediately preceding the generated `while` loop; moving *for-clause-3* to a new *block-item* placed at the end of a generated compound statement containing the body statement; and using *for-clause-2* as the controlling expression of the generated `while` loop.

```
STMT1;
for (CL1; CL2; CL3)
  STMT2;
CONT;

// is rewritten as

STMT1;
CL1;
while (CL2) {
    STMT2;
    CL3;
}
CONT;
```

The body of the generated `while` loop is then re-examined for rewriting. As per 7.1, the scope of any identifiers introduced in the `for`-clauses shall not be affected by this rewrite.

NOTE it is the implementation's responsibility to either rename identifiers or bind them to an entity before the rewriting step, so that scope is not affected.

```
// ...ultimately producing a result equivalent to

STMT1; // executed once, not part of the recursion
CL1;   // also only once

return [] (Locals ...ls) {  // <-- recursion jumps to here
    if (!(CL2)) {           // not-branch first
        [return] CONT;      // (CONT is already a return)
    } else {
        STMT2;
        CL3;
        return _Recur (ls); // repeat loop
    }
} (...);
```

In all three cases the resultant encapsulating function is parameterized by all objects in scope that are used in the statement and its continuation (including any objects declared by `CL1`).

7.2.6 return statement

This section applies to both the `return` statement and, if supported, to the `return goto` statement.

Feature: terminate execution of the function and produce its final result value.

Description: a return statement has no local continuation.

Correspondingly, a continuation will be a return statement building up the exit value from the containing function.

When a return statement appears in a position that implies that a continuation is appended to it, the continuation is discarded:

```
if (COND) {
    return V;
}
CONT;

// would normally be rewritten as

if (COND) {
    return V;
    CONT; // <-- but this is discarded
} else {
    CONT;
}

// so instead:

if (COND) {
    return V;
} else {
    CONT;
}
```

No rewrite operations are applied to the return statement itself.

7.2.7 `break` statement

Feature: terminate execution of a loop or `switch` and move to the next statement.

Description: a `break` statement is rewritten by discarding it and replacing it with a copy of the continuation of its associated loop or selection statement. The associated statement is the one identified by the description of the `break` statement in C23 section 6.8.7.4.

```
while (COND1) {
    STMT1;
    if (COND2)
        break;
}
CONT;

// is rewritten equivalently to

while (COND1) {
    STMT1;
    if (COND2)
        CONT;
}
CONT;

// and then rewritten (per 7.1.3) as

return [] (Locals ...ls) {
    if (!(COND1) {
        CONT;
    } else {
        STMT1;
        if (COND2)  // this will be rewritten in a subsequent step
            CONT;
        return _Recur (x);
    }
} (x);
```

The continuation has already been rewritten by this point.

Any existing continuation that would otherwise apply to the `break` statement is discarded:

```
while (COND) {
    STMT1;
    break;
    STMT2; // inaccessible
}
CONT;

// is rewritten equivalently to

while (COND) {
    STMT1;
    CONT;
    // STMT2 discarded
}
CONT;
```

7.2.8 `continue` statement

Feature: terminate execution of one iteration of a loop and move on to the next iteration.

Description: a `continue` statement is rewritten by discarding it and replacing it with a copy of the *generated local continuation* for the body of its associated loop or selection statement. The associated statement is the one identified by the description of the `continue` statement in C23 section 6.8.7.3.

NOTE the generated local continuation is defined in 7.1.4 as the continuation inserted during rewriting of the loop, as `return _Recur ( ....`

```
while (COND1) {
    STMT1;
    if (COND2)
        continue;
    STMT2;
```

```
}
CONT;

// is rewritten (per 7.1.3) as

return [] (Locals ...ls) {
    if (!(COND1) {
        CONT;
    } else {
        STMT1;
        if (COND2)  // this will be rewritten in a subsequent step
            return _Recur (x);
        STMT2;
        return _Recur (x);
    }
} (x);
```

Any existing continuation that would otherwise apply to the `continue` statement is discarded.

7.2.9 `goto` statement

Feature: unconditional jump to a labelled statement.

Description: the `goto` statement shall jump to a statement that exists within the continuation of its immediately enclosing statement, as defined by 7.1 (therefore, not including statements only reachable through the *generated local continuation* of a loop body), or within the continuation that would be the `goto` statement's continuation if it was not a jump statement.

NOTE this means that `goto` does not describe a "backwards jump".

The `goto` statement is rewritten by discarding it and replacing it with a copy of the continuation named by the label. The named continuation has already been rewritten by this point.

```
void foo (int n) {
    if (n)
        goto skip;
    CONT1;
skip:
    CONT2;
}

// is rewritten as

void foo (int n) {
    if (n)         // (mid-process of rewriting the if)
        CONT2;     // this already leads to a final return
    CONT1;
    CONT2;
}
```

Any existing continuation that would otherwise apply to the `goto` statement is discarded.

7.2.10 `switch` statement

Feature: jump to a labelled nested statement depending on the value of a controlling expression.

Description: The `switch` statement is rewritten as follows:

First, the *secondary-block* is rewritten.

Each `case` and `default` label within the *secondary-block* and associated with the `switch` per the description in C23 section 6.8.5.3 is then identified and replaced by a generated named label.

The list of `case` and `default` statements is used to build an equivalent `if` statement:

```
switch (EXPR) {
case 1:
    STMT1;
case 2:
    STMT2;
default:
    STMT3;
case 5:
    STMT4;
}

// is rewritten as

if      ((EXPR) == 1) goto @case_1;   // goes to STMT1
else if ((EXPR) == 2) goto @case_2;   // goes to STMT2
else if ((EXPR) == 5) goto @case_5;   // goes to STMT4
else                  goto @default;  // goes to STMT3
```

The `switch` statement is removed and replaced by this `if` statement, which is then is then re-examined for rewriting.

NOTE the rewritten statements within the discarded `switch` will be selected to replace the `goto` statements in a subsequent step, so they are not completely discarded.

NOTE per 7.1 the label targets remain valid and identify continuations to replace the added `goto` statements, even though they are not present in the statement tree mid-rewrite.

If there is no `default` label associated with the `switch`, an `else` branch is not added to the generated `if`:

```
switch (EXPR) {
case 1:
    STMT1;
case 2:
    STMT2;
case 5:
    STMT3;
}

// is rewritten as

if      ((EXPR) == 1) goto @case_1;   // goes to STMT1
else if ((EXPR) == 2) goto @case_2;   // goes to STMT2
else if ((EXPR) == 5) goto @case_5;   // goes to STMT3
```

If there are no `case` labels and only a `default` label associated with the `switch`, the `if` is not generated and the single `goto` for the `default` label replaces it:

```
switch (EXPR) {
    struct SomeTag { ... };
default:
    STMT1;
}

// is rewritten as

goto @default;  // SomeTag is defined to still be in scope in the continuation
```

If there are no `case` or `default` labels associated with the `switch`, it is discarded without any replacement.

7.3 Detailed changes to ISO/IEC 9899:2024 plus Clause 6

The modifications are ordered according to the clauses of ISO/IEC 9899:2024 to which they refer. If a clause of ISO/IEC 9899:2024 is not mentioned, no changes to that clause are needed. New clauses are indicated with (NEW CLAUSE), however resulting changes in the existing numbering are not indicated; the clause number *mm.nn***a** of a new clause indicates that this clause follows immediately clause *mm.nn* at the same level. Bolded text within an existing clause is new text.

These changes assume the changes presented in 6.3 have been applied.

Modify 6.7.5 “Function specifiers”:

Instead of the changes to 6.7.5 specified in 6.3, add the following paragraphs after paragraph 8:

A constexpr function shall not have void return type.

A constexpr function shall return a value.

(if the ***`defer`*** *statement specified by TS 25755 has been implemented)* A constexpr function shall not register any deferred operations to be executed on exit from any scope within its definition.

A constexpr function shall not contain any specification of a variably-modified type, apart from the declaration of parameters declared with an array type before adjustment,[footnote1)] except within the operand of `sizeof` or `alignof`.[footnote2)]

footnote1) this array type is converted to an undimensioned pointer type by adjustment as described in 6.7.7.4, and is supported even when `__STDC_NO_VLA__` is defined equal to 1; no VLA or variably-modified type is defined by such a parameter.

footnote2) therefore if a variably-modified type appears in the operand of a typeof operator, this typeof operator will ultimately appear within the operand of `sizeof` or `alignof`.

No expression in a constexpr function shall evaluate the address of any object, except via the implicit conversion of an array expression to a pointer to its first element during value conversion. No expression within the definition of a constexpr function shall dereference any pointer with the unary * operator, except within an unevaluated operand of the `sizeof`, `alignof` or typeof operators. No evaluated expression in a constexpr function shall contain a call to a function declared without the `constexpr` specifier.

If the definition of a constexpr function contains a `goto` statement, the label named by the identifier shall not appear before the `goto` statement within the function definition.

No expression within the definition of a constexpr function shall contain any subexpression that would require any identifiers appearing as operands to be *modifiable lvalues*, except within an unevaluated operand of the `sizeof`, `alignof` or typeof operators; or as the left-hand operand of an assignment expression, increment, or decrement, that (ignoring parentheses and generic selection) appears as the top-level operator of an expression statement.

No expression within the definition of a constexpr function shall read the value of any member of a union other than the member last used to store a value.

No declaration within a constexpr function shall define an object with a volatile-qualified or atomic type. No expression in a constexpr function shall access an lvalue with volatile-qualified or atomic type.

Within an assignment, increment, or decrement expression, the lvalue operand shall designate an object defined in the containing function with automatic storage duration, or a sub-object of such an object accessed by means of the `.` or `[]` operators. The lvalue operand shall not designate an element of an array passed as a pointer parameter to the containing function, or any sub-object of such an element.

If any declaration of a function has a `constexpr` specifier, then all of its declarations shall contain a `constexpr` specifier.

Add a further definition to 6.10.10.4 “Conditional feature macros”:

`__STDC_CONSTEXPR_EXTENDED_TS__` The integer constant 1, intended to indicate conformance to the extended specifications for constexpr functions in TS 25007.

Index