TS Draft
WG14 3205

Title: TS draft: C - Extensions to support pure functions
Author, affiliation: Alex Gilding, Perforce
Date: 2023-12-14
Proposal category: Technical Specification
Target audience: Compiler/tooling developers

Abstract

This Technical Specification aims to lay out common practice guidelines for features related to extended pure 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 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 feature area is largely based on C feature proposal document N2917, which builds upon the features described in N3018 that were accepted for standardization in C23.

The second feature area 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. This feature area is largely based on C feature proposal document N2920.

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 9899:2023.

Each clause in this Technical Specification deals with a specific topic. The first subclauses of clauses 4 and 5 contain a technical description of the features of the topic. These subclauses provide an overview but do not contain all the fine details. The last subclause 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.

Additional explanation and rationale are provided in the Annexes.

2. References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:2023 – Programming languages – C (draft n3096)

C++11 (draft n3337)

C++20 (draft n4868)

R7RS Scheme

n2917 The `constexpr` specifier, v2

n2920 Tail-call elimination, v2

n3018 The `constexpr` specifier for object definitions, v7

3. Conformance

This Technical Specification presents in two separate clauses specifications for two, in principle independent, sets of functionality (clause 4: extended constant expressions, clause 5: tail-call elimination). 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 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.).

4. Extended Constant Expressions

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 behaviour 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 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.

4.1 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 feature of C23 and are described in ISO/IEC 9899.

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.

Within a `constexpr` function, its parameters are also treated as `constexpr` objects with the values of their arguments, in the limited context of its return expression (forward reference: 4.1.2).

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; 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 behaviour. An expression containing undefined behaviour in an evaluated branch is not a constant expression for any purposes.

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.

4.1.1 `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 assignment if they are members of a containing structure), 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, before parameter adjustment. 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 dimension of the array.

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, 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.

4.1.2 `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). 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)

...in addition to exactly one return statement which evaluates a constant-expression according to the modified rules of this specification. 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).

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++'s slightly different `inline` rules *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. 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) 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.

When the name of an array defined with the `constexpr` storage class is passed as an argument to a `constexpr` function, if (and only if) the corresponding parameter is declared with the type of an array of the same element and the same or smaller constant dimension (before adjustment), that parameter is treated as a `constexpr` array within the function return expression according to the rules in 4.1.1, and may be indexed by integer constant expressions that make use of the values of any other parameters.

Further discussion of the rationale for adopting the C++11 ruleset (as far as relevant to C) can be found in WG14 document n2917. Of note is that this ruleset does not require any interpreter semantics to be added to the C translator, as all value uses are pure and all function calls can be

expanded essentially like macros, by direct, scope-aware, substitution of the result expression into the call site, to be evaluated using the existing constant evaluation engine.

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.

4.2 Detailed changes to ISO/IEC 9899:2023

Changes are relative to Working Draft n3096, the last public draft of C23 before release.

The modifications are ordered according to the clauses of ISO/IEC 9899:2023 to which they refer. If a clause of ISO/IEC 9899:2023 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 new clause indicates that this clause follows immediately clause *mm.nn* at the same level.

Add two new bulleted entries to 5.2.4.1 “Translation limits”, with values matching those in C++20:

> **- 512 nested `constexpr` function invocations**
> **- 1048576 nested *constant expressions* within the evaluation of a single *constant expression***

Modify 6.3.2.3 “Pointers”:

Modify paragraph 3:

> An integer constant expression with the value 0, or such an expression cast to type `void *`, the predefined constant `nullptr`, **or such an expression returned from a constexpr function,** is called a *null pointer constant*.

Add a forward reference:

> equality operators (6.5.9), **function specifiers (6.7.4),** integer types capable of holding object pointers (7.20.1.4)

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. The function to call shall be defined in the translation unit before the evaluation of the outermost constant-expression containing the (possibly-nested) call.**

Add a new sentence to paragraph 7 explaining that array elements can be constants:

> **An expression accessing an element of an array using the subscript operator is a *named constant* if the pointer-typed operand is an identifier that was declared with the `constexpr` storage-class specifier and complete array type[footnote), and the**

subscript index is an integer constant expression greater than or equal to zero and less than the declared length of the array type.

footnote) before any parameter type adjustment.

Add a new paragraph after paragraph 7:

Within the return expression of a constexpr function and within the array size expressions of any parameters (but not within any declarations within the function body), the function parameter identifiers are treated as though they were declared with the `constexpr` storage-class specifier and function as *named constants*.

Paragraph 8, include function calls returning integer values:

An integer constant expression 132) shall have integer type and shall only have operands that are integer constants, named and compound literal constants of integer type, character constants, `sizeof` expressions whose results are integer constants, `alignof` expressions, and floating, named, **calls to constexpr functions that return a value with integer type,** or compound literal constants of arithmetic type that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the typeof operators, `sizeof` operator, or `alignof` operator.

Paragraph 8, include function calls:

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.** Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the typeof operators, `sizeof` operator, or `alignof` operator.

Paragraph 12, modify dereferencing rules to allow member/element access:

but the value of an object shall not be accessed by use of **the `*` or `->`** operators **footnote1)**. **A value may be accessed by use of the subscript operator if its pointer-typed operand is an address constant designating a `constexpr` array with static storage duration footnote2).**

footnote1) therefore a `constexpr` function may receive an address constant as an argument, but may not dereference it using `*`.

footnote2) to designate a `constexpr` array, the array must have already been defined and initialized before the subscript operator expression is evaluated, because a `constexpr` object declaration is a definition; or be a parameter of a constexpr function declared with complete array type.

Add a new paragraph after paragraph 12:

An address constant may be returned from a `constexpr` function and if so remains an address constant in the calling context.

Add a forward reference:

array declarators (6.7.6.2), **function specifiers (6.7.4),** initialization (6.7.9).

Modify 6.7.1 "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 (6.7.8)**, function specifiers (6.7.4)**.

Modify 6.7.4 "Function specifiers":

Paragraph 1, add the `constexpr` specifier:

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

Add four new paragraphs after paragraph 3:

> **A function declared with the `constexpr` function specifier shall not have `void` return type.**
>
> **A function defined with the `constexpr` function specifier shall return a value, and shall contain only:**
>
> - **null statements**
> - **static assertions**
> - **typedef and tag declarations**
> - **definitions of objects with the `constexpr` storage class**
> - **a single return statement, which evaluates a constant-expression according to the rules of 6.6, treating the parameters of the function as *named constant* expressions within the `return` footnote).**
>
> **footnote) the parameters are not treated as *named constants* within any static assertions or declarations, only within the `return` statement.**
>
> **If any declaration of a function has a `constexpr` specifier, then all of its declarations shall contain a `constexpr` specifier.**
>
> **Any array types specified within the prototype or body of a constexpr function shall not be variably-modified, except that the expression specifying the array size of a parameter may treat the function parameters as constant primary identifier expressions footnote).**
>
> **footnote) implying that for calls with constant arguments the declared size of the array parameter is statically known even if it depends on the value of other parameters.**

Add five new paragraphs after paragraph 8:

A function declared with a `constexpr` function specifier is a constexpr function. A call to a constexpr function identifier with arguments that are all constant expressions is itself a constant expression of the same category as its return expression, and may be called in contexts such as static assertions or initialization of objects with static storage duration after it has been defined.

A constexpr function whose return expression is an integer constant expression and has an integer return type returns an integer constant expression, and a constexpr function whose return expression is a null pointer constant and has the same return type returns a null pointer constant.

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

A constexpr function does not modify any state, or observe any state outside of its own argument values. Any constexpr function may be declared with the `[[unsequenced]]` attribute and fulfils its requirements.

A constexpr function is implicitly also an inline function.

Add a NOTE:

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 definition, 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 a Recommended Practice:

Recommended Practice

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.

Add an example:

EXAMPLE 2 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); //
}
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
```

Add a forward reference:

> Forward references: function definitions (6.9.1)**, Attributes (6.7.12)**.

No modifications are made to the Standard Library.

5. Tail-Call Elimination

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), 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, which mandates its support.

5.1 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 (after stripping parentheses) operand to a `return goto` statement.

Although most of the research on tail-calls 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` subtype of the `return` statement, is needed, as an attribute cannot impose the required change in behaviour 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 behaviours 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 be 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.

5.1.1 Calling conventions and ABI

No new ABI is introduced by this feature. Although the a 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.

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; **however**, the TU would also need to be recompiled in order to make use of the explicit tail call syntax, as the `return goto` statement is not present in C23. (From outside the TU, all entry to such a function would be by “far call” anyway and therefore the ABI presented to other TUs would not change.) It was not clear that this presents a blocking issue in practice.

Since a “near call” implies information from the function definition is visible to the compiler, no annotation of functions ending in a `return goto` statement is required for them to rebuild such TUs correctly. The lack of annotation or any type-based distinction for callers ending in a proper *tail-call* is preserved and the detail remains semantically private.

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 (signatures that allow “return to next”). It is assumed that users of such functions actively *want* their ABI to change and become seamless, once that becomes possible.

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 will 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 cannot be stored in the same pointer variable, the compiler must 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 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 behaviour to add and covers all cases where the call would be difficult to translate.

5.2 Detailed changes to ISO/IEC 9899:2023

Changes are relative to Working Draft n3096.

The modifications are ordered according to the clauses of ISO/IEC 9899:2023 to which they refer. If a clause of ISO/IEC 9899:2023 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 new clause indicates that this clause follows immediately clause *mm.nn* at the same level.

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 **footnote)**.)

footnote: unless entry to the call explicitly terminates execution of the current block with `return goto`.

Add a forward reference to (new section) 6.8.6.4.1 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 (6.8.6.4.1),** initialization (6.7.9), statements (6.8), effective type (6.5).

Add a new footnote to 6.5.2.2 "Function calls" paragraph 8, immediately after 106:

with respect to the execution of the called function. 106) **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 6.8.6 "Jump statements" paragraph 1:

jump-statement:
`goto` identifier `;`
`continue ;`
`break ;`
`return` *expression* opt `;`
`return goto` ***postfix-expression*** **`;`**

(NEW CLAUSE) Add a new section, 6.8.6.4.1 "The `return goto` statement":

6.8.6.4.1 The `return goto` statement

Constraints

The expression shall be a function call, optionally enclosed in parentheses. No other form of expression is permitted.

Within the function call expression, the *postfix-expression* designating the function to call shall have a type compatible with the type of a pointer to the function whose definition contains the statement.

Neither the function to call nor the containing function shall have a parameter list that terminates with an ellipsis.

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 `return` statement.

Semantics

A `return goto` statement is a special case of the `return` statement which terminates execution of the current function and then passes direct control to the function specified in the function call that constitutes its operand expression.

The value of that function call 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 return directly to the current function's caller.

footnote) and the implicit copy of any argument values to the 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.

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

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.

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 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 (&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.

No modifications are made to the Standard Library.