```
Proposal for C23
WG14 2917

Title:                The `constexpr` specifier
Author, affiliation: Alex Gilding, Perforce
Date:                 2022-01-25
Proposal category:    New features
Target audience:      Compiler/tooling developers, library developers,
application developers
```

# Abstract

C++ has supported compile-time evaluation of first-class functions for over ten years, while C is still limited to using second-class language features in compile-time contexts. This puts C at a significant disadvantage in terms of being able to share the same features between runtime and compile-time, and in being able to assert truths about the program at compile time rather than waiting to assert in a runtime debug build.

`constexpr` provides the ability for a C program to call a restricted set of functions at compile time while leaving them first-class language citizens.

# The `constexpr` specifier

Reply-to: Alex Gilding (agilding@perforce.com)
Document No: N2917
Revises Document No: N2851
Date: 2022-01-25

## Summary of Changes

### N2917

- recursion limits; no UB in C++; no new ODR; no call before definition; linkage; initializer order
- wording for function definitions, avoid VLA side effects
- wording for compound literals
- split wording for different kinds of constant expression and propagate kind; add wording to null pointer constant
- community comments, implementability

### N2851

- original proposal

## 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.

We propose to add a new (old) specifier to C, `constexpr`, as introduced to C++ in C++11. We propose to add this specifier separately to objects and functions, and to intentionally keep the functionality minimal to avoid undue burden on lightweight implementations.

## Rationale

Because C limits initialization of objects with static storage duration to constant expressions, it can be difficult to create clean abstractions for complicated value generation. Users are forced to use macros, which do not allow for the creation of temporary values and require a different coding style. Such macros - especially if they would use temporaries, but have to use repetition instead because of the constraints of constant expressions - may also be unsuitable for use at runtime because they cannot guarantee clear evaluation of side effects. Macros for use in initializers cannot have their address taken or be used by linkage and are truly second-class language citizens. A user is obliged to repeat themselves and provide both a macro and a function (probably just deferring

operation to the macro internally, but still unclear and verbose) if they wish for the same callable entity to be used in both a static context and a runtime context.

The same restriction applies to `static_assert`: a user cannot prove properties about any expression involving a function call at compile-time, instead having to defer to runtime assertions. If a constant function could be called at compile-time, a release-mode build would be able to bake more testing of values and value-generation directly into the build step rather than relying on a separate debug configuration and test runs of the full program.

C does provide enumerations which are marginally more useful than macros for defining constant values, but their uses are limited and they do not abstract very much; in practice they are only superior in the sense that they have a concrete type and survive preprocessing. Enumerations are not really intended to be used in this way.

In C++, both objects and functions may be declared as `constexpr`, allowing them to be used in all constant-expression contexts. This makes function calls available for static initialization and for static assertion-based testing.

In the case when a `constexpr` function is not visible, it may also provide useful information to an optimizer by communicating that its return is not affected by the rest of the program state, which allows multiple calls with identical arguments to be folded into single calls even without a visible definition. (In practice this is less interesting as the definition will be visible most of the time.)

The subset of headers which are able to be common between C and C++ is also increased by adding this feature and strictly subsetting it from the C++ feature. Large objects can be initialized and their values and generators asserted against at compile time by both languages rather than forcing a user to switch to C++ solely in order to get compile-time assertions.

# Proposal

We propose adding the new keyword `constexpr` to the language and making it available:

- as a storage-class specifier for objects
- as a function specifier.

We also propose relaxing the constant-expression rules to allow access to aggregate members when the object being accessed is declared as a `constexpr` object and (in the case of arrays) the element index is an *integer constant expression*.

A scalar object declared with the `constexpr` storage class specifier is a constant. It must be fully and explicitly initialized according to the static initialization rules. It still has linkage appropriate to its declaration and it exist at runtime to have its address taken; it simply cannot be modified at runtime in any way, i.e. the compiler can use its knowledge of the object's fixed value in any other constant expression.

There are no restrictions on the type of an object that can be declared with `constexpr` storage duration (because all C object types are completely trivial, in C++ terms). It does not make sense to use any of the currently provided Standard qualifiers on a `constexpr` object but it is not necessary to impose this as a constraint either (since it cannot be modified: `const` is implicit, `_Atomic` is unnecessary, and `volatile` does not hurt because the object still exists to reload at

runtime, but won't do anything either). Other qualifiers may be introduced at a later time that might hold more meaning for these objects.

A function declared with the `constexpr` function specifier is subject to stricter restrictions, taken from the quite restrictive set of (relevant) rules used by C++11. The function body may only contain:

- null statements (plain semicolons)
- `static_assert` declarations
- `typedef` declarations

...in addition to exactly one return statement which evaluates a constant-expression according to these modified rules. The function must return a value (or it is useless).

A `constexpr` function is implicitly also an `inline` function, allowing it to be defined in a header.

A `constexpr` function, called 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. The address of a `constexpr` function may be taken and used as any other function pointer; this does not preserve the `constexpr` specifier.

We currently propose to stay entirely within (the relevant C subset of) the C++11 restrictions, for implementation-focused reasons. This imposes a much lighter burden on implementations, which must already provide expression evaluators, than extending the ruleset to match C++14 and later, which start to require statement evaluation and mutable local variables, generally demanding the beginnings of a true interpreter built into the compiler.

We do not propose changing the meaning of the `const` keyword in any way (this differs between C and C++) - an object declared at file scope with `const` and without `static` continues to have external linkage; an object declared with static storage duration and `const` but not `constexpr` is not considered any kind of constant-expression, barring any implementations that are already taking advantage of the permission given in 6.6 paragraph 10 to add more kinds of supported constant expressions.

The difference between the behaviour of `const` in C and in C++ is unfortunate but is now cemented in existing practice and well-understood. We would oppose changing that now.

We do not propose changing the meaning of the implied `inline` specifier on a `constexpr` function to match C++'s `inline`. A C `constexpr` function that wants to provide an externally linkable definition should use `extern inline` the same as current C `inline` functions do.

No modifications are currently proposed for Section 7 as the Standard Library was not developed with the `constexpr` concept in mind. The specifier can be added on an individual basis to functions once the feature is available language-wide.

# Alternatives

C currently has only one class of in-language entity that can be defined with a value and then used in a constant context, which is an enumeration. This is limited to providing a C-level name for a

single integer value, but is extremely limited and is a second-class feature closer to macro constants than to C objects. These cannot be addressed and also cannot be used to help much in the abstraction of function-like expressions.

GCC provides two non-standard attributes, `const` and `pure`, that are similar to this proposal. These attributes mostly communicate intent to calling code rather than impose restrictions on the function body itself. They are only applied to functions and do not substantially change the way C features can be used in expressions. `const` is closer to `constexpr` as it prevents the function from reading mutable state (`pure` merely says the function will not modify external state).

n2539 "Unsequenced functions" by Alepin and Gustedt brought a number of proposed `[[attributes]]` that could annotate functions as having one of five levels of "unsequence", from full referential transparency (`const`) through to simply not leaking resources (`noloeak`). The stricter levels were a direct attempt to standardize the GNU attributes.

Unfortunately an attribute-based solution does not provide the user-facing functionality of being able to simply use more complicated expressions within initializers and static assertions, because a standard C attribute *must* leave the program correct when it is removed. If the `constexpr` nature of a function depends on an attribute, ignoring the attribute would change whether a program is valid at all. Therefore these attributes are mostly in aid of a slightly different goal of communicating more intent to the compiler about which external calls can be reordered or optimized away; they do not change the code a user can directly write.

Some enhancements in the above proposal are also worth separate discussion:

- the C++11 rules do not allow local variables to be declared within a `constexpr` function. This is an overly-strict restriction if such definitions were always `const` themselves (but not `constexpr`, so a local automatic could not contribute to the initialization of a local static). Allowing local variables would greatly improve the clarity of complicated expressions that might currently involve a lot of repetition. Under the C++11 rules, temporary values can only be established by recursively calling another `constexpr` function and passing them as arguments.
- under the currently-proposed rules a pointer to a `constexpr` function cannot be passed as an argument to another `constexpr` function and used at compile-time. This may be useful but is not within the proposed rules.
- allowing recursion means that evaluating a poorly-written function may fail at compile time due to an infinite loop or compiler resource exhaustion. If we disallow recursion, all `constexpr` calls would become fully inlinable. This is a strengthening of the C++11 ruleset.

We can also observe that compilers are currently free to extend the set of valid constant expressions in an implementation-defined way (6.6 p10 “An implementation may accept other forms of constant expressions”). This implies that a suitably-designed compiler can potentially feed information back from its inliner/optimizer, all the way to the type system. However, we do not consider dataflow-based optimizations relevant as prior art, as this feature needs to exist at a pre-optimization stage of compilation.

We received user feedback that it would also be useful to mandate that one or more parameters to a function always receive a constant argument. This has been proposed as another use for the `constexpr` specifier in C++, but was rejected. C++ does not fundamentally need this feature because it can use non-type template arguments instead. (C++20 `consteval` functions partially overlap here, but do not allow for only some arguments to be constant.)

We inevitably received *user* feedback that C++14-style further relaxations of the rules would be useful. This observation is true, but we received no corresponding feedback from implementers that the burden of providing it would be reasonable.

# Impact

The first question is of implementation burden. Barring recursive calls (or the possible extension to allow calls by pointer), a `constexpr` function call would be fully inlinable down to a fixed size expression tree which could then be evaluated the same way that an implementation currently evaluates a constant-expression with no function calls. Simple replacement of parameters by the argument values (as currently happens with macros used to abstract constant expressions) would be semantically correct. We consider this to be a modest implementation requirement.

## Recursion

In the initial version of this proposal we chose to disallow recursion, on the grounds that this was helpful to simple implementations by allowing them to fully inline all function calls before evaluating the expanded expression tree. This would imply a phased-evaluation of function calls vs. the expressions themselves, conceptually not too far removed from the existing macro-expansion translation phase. Further examination revealed that Stroustrup and Dos Reis started with a similar assumption, and relaxed it after further work indicated that allowing (but not mandating) recursion did not pose a problem in practice (C++ n2826).

Since the runtime C language does not mandate a minimum recursion depth either, leaving the actual succeeding nested call depth entirely down to the implementation – and importantly, allowing for *completely non-reentrant* implementations of C that do not support even one level of recursion, to still be conforming – we believe it is not necessary to state this restriction. If an implementation chooses to inline all visible `constexpr` function calls in a pass before attempting to evaluate the expanded expression trees, the depth to which it can succeed is a matter of QoI, exactly the same as in the runtime language.

Allowing recursion does not introduce the kind of complicated interpreter-like behaviour implied by requiring statements, mutable locals, readable addresses etc. introduced by C++14. A slightly more powerful implementation can interleave a macro-like expansion of the expression trees (or even token-sequences, for a super-simple push/pop evaluator that doesn't build trees) with the general expression evaluator, and can handle recursion by not expanding calls within an unevaluated ternary/logical branch just the same as it ignores the values of the expressions there (already left alone, as they might otherwise have UB).

We believe that for a classical tree-based expression evaluator, the implementation is relatively trivial. QAC's C compiler provides implementation experience integrating into such an evaluator. We believe that integration into a non-tree-based evaluator would not be substantially more difficult because in that case the call-expansion would take a very similar form to function-like macro

expansion. Implementations using more complicated, low-level/VM-like models for their constant evaluators are unlikely to have difficulty here.

We received feedback from one third-party “small compiler” implementor: the developer of ChibiCC reported that implementation would not be hard, in their opinion.

Again, we believe that since the core language does not mandate recursive calls succeed in a conforming implementation, it is safe to let the `constexpr` subset mirror this and provide recursion of depth greater than zero as a QoI feature; or not, if this is too difficult to achieve.

### C++

As above, the existing incompatibility of `const` between C and C++ is preserved because the proposal does not intend to break or change any existing C code. Code that intends to express identical constant semantics for values in both C and C++ should start using `constexpr` objects instead.

We received some repeated community feedback about whether the feature needed new ODR-style rules to control where the runtime definitions of functions reside and which TUs had visibility. We do not believe *any* new rules are required – C99’s `inline` rules have the exact same set of (non-)issues and have been adopted well in practice.

Similarly as for `const`, the existing differences between `inline` in C and C++ should be preserved for consistency across all `inline` functions defined in a C program.

This change improves C's header compatibility with C++ by allowing the same headers to make use of better compile-time initialization features. This increases the subset of C++ headers which can be used from C and does not impose any new runtime cost on any C program.

In C++ a core constant expression cannot invoke any undefined behaviour (C++20 [expr.const] paragraph [5.7]). We could add wording to this effect to C as well; but we do not consider it directly relevant to the feature proposed here, rather this would be a general improvement to all constant expressions in C, including the existing rules.

An almost-unrelated but extremely useful impact emerges from the change to make aggregate element access a constant expression: this would make it possible to statically assert that a string is a string literal (or semantically equivalent to one, at any rate), by checking the final `char` equals `'\0'` in a constant-expression context. This is useful in a number of other situations such as `printf` successors.

## Implementation Experience

There is widespread implementation experience of `constexpr` as a C++ feature.

Internally to the QAC team, we have experience fitting C++11 ruleset `constexpr` to the C constant evaluator. Our C frontend does not share this component with the C++ compiler, so we were able to compare and contrast which work was reasonable to import and which was not (i.e. we have implemented `constexpr` fully before). We felt that full C++20 ruleset `constexpr` was

completely unreasonable (probably not controversial!), but that the C++11 rules, designed to build-up from a minimalist perspective, were not difficult for a single-person team to add to a C evaluator.

We do not consider optimizer-based inlining to be direct implementation experience because it does not feed back the results of the expansion in a program-observable way. However, the technology itself may be shareable depending on the compiler architecture.

# Proposed wording

Changes are proposed against the wording in C23 draft n2731. Bolded text is new text.

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 6.5.2, “Postfix operators”:

Paragraph 1, add the `constexpr` specifier to compound literal syntax:

> *postfix-expression:*
> *primary-expression*
> *postfix-expression* `[` *expression* `]`
> *postfix-expression* `(` *argument-expression-list opt* `)`
> *postfix-expression* `.` *identifier*
> *postfix-expression* `->` *identifier*
> *postfix-expression* `++`
> *postfix-expression* `--`
> `(` ***compound-name*** `)` `{` *initializer-list* `}`
> `(` ***compound-name*** `)` `{` *initializer-list* `,` `}`
>
> ***compound-name:***
> **`constexpr`** ***type-name***
> ***type-name***

Modify 6.5.2.5, “Compound literals”:

Modify paragraph 2:

> **If the *type name* is prefixed by the `constexpr` specifier, all the expressions in the initializer list shall be constant expressions.**

Modify paragraph 3:

> A postfix expression that consists of a parenthesized type name**, optionally prefixed by `constexpr`,** followed by a brace-enclosed list of initializers is a *compound literal*.

Modify 6.2.2, “Linkages of identifiers”, paragraph 3:

> If the declaration of a file scope identifier for an object or a function contains the storage-class specifier `static`, **or the declaration of a file scope identifier for an**

**object contains the storage-class specifier `constexpr`,** the identifier has internal linkage.

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 *`, **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 paragraph after paragraph 3 explaining that aggregate elements can be constants:

**An expression accessing an element of a structure or union type is a constant if the structure or union object was declared with the `constexpr` storage-class specifier, and the accessed element was previously initialized footnote). An expression accessing an element of an array using the subscript operator is a constant if the array was declared with the `constexpr` storage-class specifier, and the subscript index is an integer constant expression designating an element within the array.**

**footnote) therefore an expression accessing a union member is not constant if it accesses a different member from the one that was initialized.**

Paragraph 6, include function calls returning integer values:

An integer constant expression 127) shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, `_Alignof` expressions, **calls to `constexpr` functions that return a value with integer type,** and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the `sizeof` or `_Alignof` operator.

Paragraph 7, add a line:

an integer constant expression **, or**

- **a structure or union object footnote) defined with the `constexpr` storage-class specifier, or returned from a call to a `constexpr` function; or a member of such a structure.**

**footnote) including compound literals.**

Paragraph 8, include function calls:

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

Paragraphs 6 and 8 do not need additional text to mention aggregate elements.

Paragraph 9, 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 postfix-expression 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.**

Add two new paragraphs after paragraph 9:

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

**A *structure constant expression* shall be a value of structure type whose elements are all constant expressions. A structure constant expression is copied by value from an initialized constant structure object, including compound literals.**

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":

Paragraph 1, add the `constexpr` specifier:

*storage-class-specifier*:
`typedef`
`extern`
`static`
`_Thread_local`

`auto`
`register`
**`constexpr`**

Add to paragraph 2:

> At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that `_Thread_local` **or `constexpr`** may appear with `static` or `extern`.

Add a new paragraph after paragraph 3:

> **`constexpr` shall not appear in the declarators of an object of pointer type, or of an aggregate containing a subobject of pointer type.**

Add two new paragraphs after paragraph 5:

> **The `constexpr` specifier is treated as a function specifier when applied to a function declaration.**
>
> **When the `constexpr` specifier is applied to an object declaration, the declaration shall be a definition.**

Add three new paragraphs after paragraph 8:

> **An object declared with the `constexpr` storage-class specifier has its value permanently fixed at translation-time. Its value can therefore be used as a constant expression of the respective kind (6.6). A constexpr integer object may be used as an integer constant expression; a constexpr arithmetic object may be used as an arithmetic constant expression; a constexpr structure may be used as a structure constant expression; as well as any other kinds of constant expression supported by the implementation footnote).**
>
> **footnote) since a pointer object cannot be defined with the `constexpr` specifier, a constexpr object cannot be used as an address constant.**
>
> **`const`-qualification of the object's type is implied.**
>
> **NOTE: an object with automatic storage duration declared with the `constexpr` storage-class specifier still has a unique address, obtainable with &.**
>
> **An object with static storage duration that is declared with the `constexpr` storage-class specifier has internal linkage.**

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 declarations**
- **a single return statement, which evaluates a constant-expression according to the rules of 6.6, treating the parameters of the function as constant primary identifier expressions within the `return` footnote).**

**footnote) the parameters are not treated as constant identifiers within any static assertions, only within the return statement and within array sizes.**

**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 constant expression specifying the array size may treat the function parameters as constant primary identifier expressions.**

Add five new paragraphs after paragraph 7:

**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 kind 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. A constexpr function whose return expression is an arithmetic constant expression and has an arithmetic return type returns an arithmetic constant expression. A constexpr function whose return expression is an address constant expression and has a pointer return type returns an address constant expression. A constexpr function whose return expression is a structure constant expression and has the same structure return type returns a structure constant expression. 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 other kinds of constant expression, consistent with any others it accepts (6.6).**

> **A constexpr function does not modify any state, or observe any state outside of its own argument values.**
>
> **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.**

# References


- C23 n2731
- C++11 n3337
- C++20 n4868
- n2539 Unsequenced functions
- (C++) n2826 “Issues with `constexpr`”
- (C++) p1063r2 “Core coroutines”
- GNU attribute const
- ChibiCC


Additional thanks to Jens Gustedt, Martin Uecker, Joseph Myers for detailed wording improvement suggestions.