```
Proposal for C23
WG14 2851

Title:                The `constexpr` specifier
Author, affiliation: Alex Gilding, Perforce
Date:                 2021-10-08
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: N2851
Revises Document No: N/A
Date: 2021-10-08

Summary of Changes

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 tighten the C++11 restrictions and prevent a `constexpr` function from calling itself recursively, for implementation-focused reasons.

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.

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.

Allowing recursion would introduce more complicated interpreter-like behaviour on the part of the compiler's constant evaluator, requiring a stack or an interleaved evaluator/term-rewriter. We believe this is an unreasonable burden on small implementations and therefore would vote to restrict the use of recursion beyond what is allowed by C++11.

Therefore, the rationale for prohibiting recursive calls is that without them, the expression tree for any constant expression can simply be fully inlined and expanded before evaluation begins, leaving a tree that can then be evaluated by current C constant evaluators with comparatively small changes. Allowing recursion would make this impossible and require at least some decision-making about control flow *after* parts of the tree had already started to be evaluated, bringing the evaluator closer to becoming a full-fledged interpreter. We consider this too demanding a request to impose on smaller compiler teams.

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.

As above, 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.

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.

Proposed wording

Changes are proposed against the wording in C2x draft n2596. Bolded text is new text.

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. **A function-call operator appearing in a constant expression shall only be a direct call to the identifier of a function declared with the `constexpr` function specifier.**

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. 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.**

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

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, add a new sentence:

> but the value of an object shall not be accessed by use of these operators. **An address constant may be returned from a `constexpr` function and remain an address constant.**

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 5:

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

Add a new paragraph after paragraph 8:

> **An object declared with the `constexpr` storage-class specifier has its value permanently fixed at compile-time. Its value can therefore be used as a constant expression (6.6). `const`-qualification of the object's type is implied. An object with automatic storage duration declared with the `constexpr` storage-class specifier still has a unique address.**

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 three new paragraphs after paragraph 3:

> **A function declared with the `constexpr` function specifier shall return a value.**
>
> **A function declared with the `constexpr` function specifier shall contain only:**
>
> - **null statements**
> - **static assertions**
> - **typedef declarations**
> - **a single return statement which evaluates a constant-expression (treating the parameters of the function as constant expressions for the purposes of the `return`).**
>
> **A function declared with the `constexpr` function specifier shall not call itself, either directly or indirectly.**

Add two 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, and may be used in contexts such as static assertions or initialization of obects with static storage duration after it has been defined.**
>
> **A constexpr function does not modify or observe any state outside of its own arguments and return value.**
>
> **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.**

References

- C23 n2596
- C++11 n3337
- n2539 Unsequenced functions
- GNU attribute const