constexpr class

Document Number: **P2350 R1**
Date: 2021-07-15
Project: ISO JTC1/SC22/WG21: Programming Language C++
Reply-to: Andreas Fertig ⟨ isocpp@andreasfertig.info ⟩
Audience: EWG

Contents

1 Introduction

The evolution of constexpr since C++11 allows us to make more and more parts constexpr. For example, [P0980R1] makes std::string constexpr. [P1004R2] does the same for std::vector. Microsoft's implementation [MSVCVector] shows that all member functions in std::vector are constexpr now. When I wrote the test implementation for [P2273R0] (Making unique_ptr constexpr) I more or less simply added constexpr to all member functions of unique_ptr.

[P1235R0] proposed to make all functions implicitly constexpr. Looking at the examples of vector and [P1235R0] there seems to be a desire to reduce declarators.

I propose to allow constexpr in the class-head, acting much like final, declaring that all member functions, including special member functions, in this class are implicitly constexpr:

Currently

```
class SomeType {
public:
  constexpr bool empty() const { /* */ }
  constexpr auto size() const { /* */ }
  constexpr void clear() { /* */ }
  // ...
};
```

With proposal

```
class SomeType constexpr {
public:
  bool empty() const { /* */ }
  auto size() const { /* */ }
  void clear() { /* */ }
  // ...
};
```

2 Implementation

This proposal was implemented in a fork of LLVM/Clang from the author [GHUPImpl]. The change was small and easy to apply.

3 The design

The goal is to use the existing model of final and apply it to constexpr. This reduces the noise resulting from entirely constexpr-classes as we have it now.

3.1 What about out-of-line definitions?

This proposal does not change how out-of-line definitions of constexpr member functions work. They continue to work the same way as if someone puts constexpr directly at the member function. The out-of-line definition will not compile.

3.2 What about a member function that already carries constexpr?

Well, doing things twice to be sure never hurts. The member function will be constexpr in a constexpr class regardless of whether it is declared constexpr again at member function level.

3.3 Do we need constexpr(false)?

I don't know. Feel free to bring use-cases.

My current answer is: no. If we see a constexpr class not only as a noise reduction in reading and writing but also as a promise "you can use this entire class in a constexpr-context", disabling the constexprness of certain member function makes this promise weak.

3.4 What about friend?

A friend declaration is different. Such a declaration is only in the namespace of a class but isn't a member of that class. On the reflector Ville Voutilainen provided a good example that even in a constexpr class we might have a friend declaration for an ostream operator [ml16332], which cannot be constexpr.

Therefore, this paper proposes that friend declaration are uneffected of a constexpr class. They remain as they are and need to be marked constexpr even in a constexpr class.

3.5 What about static member functions?

By this proposal static member functions get implicitly marked constexpr in a constexpr class.

3.6 What about inheritance?

Consider the following examples:

```
struct BaseCxpr constexpr {
    int foo() { return 42; }  // this member function is constexpr
};

struct DerivedA : BaseCxpr {
    int bar() { return 21; }  // this member function is _not_ constexpr
};


struct Base {
    int foo() { return 42; }  // this member function is _not_ constexpr
};

struct DerivedB constexpr : Base {
    int bar() { return 21; }  // this member function is constexpr
};
```

Listing 3.1: constexpr class and inheritance

In the case of DerivedA, where a class derives from a constexpr class, only the member functions of the constexpr base class are constexpr. There is no constexpr inheritance. It seems to constrain the design space of classes too much if only constexpr classes can derive from constexpr classes.

In the case of DerivedB, where the derived class is marked as constexpr, but the base class isn't, this proposal makes all member functions of the derived class constexpr while those of the base class remain as they are. `constexpr` for member functions explicitly marked `constexpr` in the base class and non-`constexpr` for all the others.

3.7 What about a forward declaration?

Consider this:

```
struct Forward constexpr;
```

Listing 3.2: constexpr class and forward declaration

Analogous to `final`, the above is only a forward declaration that cannot have a specifier. Hence, the code above is ill-formed by this proposal.

The same goes for class templates or specializations of class templates. Only the specialization marked as `constexpr` does have all member functions implicitly `constexpr`. All other don't.

3.8 Is adding or removing `constexpr` from the class-head a breaking change?

Say we have a class before this proposal, and after this proposal, the class author adds `constexpr` in the class-head, is this a breaking change? The short answer is no. The longer is it depends. By adding `constexpr` in the class-head *all* member functions of a class become `constexpr`. If this class had non-`constexpr` member functions before this change, then users can observe a behavioral change. However, this change is equal to adding `constexpr` to all the member functions of a class manually, what we have done in [P1004R2] to `std::vector`. This was not considered a breaking change, nor an ABI change.

3.9 Can this be solved with metaclasses?

Another question that came up is, can this feature be implemented with metaclasses. One idea is to provide such a facility with the STL. [MCSrc] lists a possible implementation that was shown in a Twitter discussion [MCSrcTweet].

While a constexpr class is implementable with the current state of metaclasses, it doesn't seem like the right tool for the job. A `constexpr class` is something simple and generic. There is no need to let the compiler generate something for us. The combination of such a metaclasses library part with other metaclasses elements, like promising shape example [P0707R4], is unclear.

3.10 Syntax choices

We have a couple of different syntax options:

```
class D constexpr : B {};  // A
class constexpr D : B {};  // B
class D : B constexpr {};  // C
constexpr class D : B {};  // D
```

A seems natural. `final` would be right of `constexpr`: `constexpr final`.

B seems a bit confusing because its before the class name. The question is does it go before or after attributes.

C seems very confusing. It creates the impression that `constexpr` applies to the base class.

D is ambiguous. We already have `constexpr class D{} d`.

This paper proposes syntax **A**.

3.11 Order of the specifiers

This paper proposes to make `final` the rightmost specifier and fill in `constexpr` to the left. The reasons are that with just `constexpr` even with a potential `consteval` the alternating freedom seems unnecessary. Teaching will be more consistent. Second, it looks as it makes the implementation easier. Currently, Clang does a scan after the class name for either a colon or an opening curly braces and checks whether the characters found are the `final` specifier. With the flexibility of placing the two specifiers both ways, this parsing gets more complex.

4 Other parts of the language

The ability to list other specifiers like `noexcept` is something that comes up with this proposal.

4.1 What about `consteval`

For consistency reasons, `consteval` should be allowed like `constexpr`.

If `consteval` is allowed as well, there are more questions to answer. It seems to make sense to allow only one of both in the class-head. Now assume a class is marked `constexpr`:

```
class SomeType constexpr {
public:
  bool empty() const { /* */ }
  // ...
};
```

Do we like to allow that a member function can be marked `consteval` and those overriding `constexpr`:

```
class SomeType constexpr {
public:
  bool empty() const { /* */ }
  // ...
  consteval bool whatheverFun() { /* */ }
};
```

The same goes the other way around. Assume we have a `consteval` class, should it be allowed that a member function can be *down-grade* to `constexpr`?

4.2 What about noexcept

noexcept acts differently than constexpr or final. Should I, as a developer, do something that is not allowed in, for example, a constexpr context the compiler gives me an error. Should I invoke a throwing function in a noexcept member function, I end up with a run-time error. It seems less desirable to me to create implicit noexcept member functions.

Another angle here are out-of-line definitions. If a full noexcept-class adds the implicit noexcept to all in-class definitions, what about out-of-line definitions? Should the also be implicitly noexcept? Should such out-of-line definitions need to be attributed with noexcept?

On the reflector, Giuseppe D'Angelo mentioned QT's Point and std::complex as examples for noexcept data structures. A quick check revealed that both data structures seem not to throw exceptions, but even std::complex is not marked noexcept in the standard. The assumed reason for them not have been marked noexcept in C++11 is that adding or removing noexcept is an observable change. If we have two functions where one is marked noexcept, and the other isn't, the typeid of them is different:

```cpp
#include <cassert>
#include <typeinfo>

void f1();
void f2() noexcept;

int main() {
    assert(typeid(f1) == typeid(f2));
}
```

Listing 4.1: Comparison of the typeid of two functions with and without noexcept.

This paper does not propose to add noexcept as a specifier in the class-head.

4.3 What about const

Another thing that could be imaginable is to have const in the class-head, declaring all member functions in a class implicitly const. This proposal does not propose this. If there is a desire for it, a dedicated proposal seems best.

In general const is different because we can have out-of-line definitions which are explicitly marked const to distinguish them from the non-const overload. A const-only class would have only const member functions, making this issue simpler, but regarding teachability and readability, dropping the const from these functions does create a new kind that seems not desirable.

This paper does not propose to add const as a specifier in the class-head.

4.4 What about override

An override class where all member functions override those in a base class would at least solve the situation with an unwanted non-virtual destructor in the base class.

This paper does not propose to add override as a specifier in the class-head.

4.5 What about free functions?

Free functions are an interesting question. While with this proposal, the noise from `constexpr`'fying entire classes is reduced, we also have a lot of cases where many free functions are `constexpr`. One example is [P1645R1], which made more algorithms `constexpr`.

One approach here can be a `constexpr` namespace like below.

```
namespace constexpr {
    bool Fun() { /* */ } // this function is constexpr
    bool Run() { /* */ } // this function is constexpr
}
```

This paper does not propose a `constexpr` namespace. If something like this is desirable, the author is open to bring another paper dedicated to such a feature.

5 Proposed wording

This wording is base on the working draft [N4885].

The wording does not include changes to STL containers. If this is desired, the author believes that it requires a new paper targeting LEWG.

Change in **[dcl.constexpr]** 9.2.5:

1 The `constexpr` specifier shall be applied only to the definition of a variable or variable template ~~or~~ , the declaration of a function or function template , or the definition of a class or class template. The `consteval` specifier shall be applied only to the declaration of a function or function template. ...

2 A `constexpr` or `consteval` specifier used in the declaration of a function declares that function to be a *constexpr function*. Further, the constexpr specifier used as a class-prop-specifier in a class definition (11.1) declares all direct member functions of that class to be constexpr functions. A function or constructor declared with the `consteval` specifier is called an *immediate function*. A destructor, an allocation function, or a deallocation function shall not be declared with the `consteval` specifier.

Change in **[class.pre]** 11.1:

class-head:

class-key attribute-specifier-seq$_{opt}$ *class-head-name* *class-prop-specifier*$_{opt}$ *class-virt-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key attribute-specifier-seq$_{opt}$ *base-clause*$_{opt}$

class-head-name:

nested-name-specifier$_{opt}$ *class-name*

class-prop-specifier:

`constexpr`

class-virt-specifier:
    `final`

Add after p5 in **[class.pre]** 11.1:

6 If a class is marked with the *class-virt-specifier* `final` and it appears as a *class-or-decltype* in a *base-clause* (class.derived), the program is ill-formed. Whenever a *class-key* is followed by a *class-head-name*, the *identifier* `final`, and a colon or left brace, `final` is interpreted as a *class-virt-specifier*. [*Example*:

```
struct A;
struct A final {};      // OK: definition of struct A,
                        // not value-initialization of variable final

struct X {
 struct C { constexpr operator int() { return 5; } };
 struct B final : C{};  // OK: definition of nested class B,
                        // not declaration of a bit-field member final
};
```

– end example]

7 [*Note*: The class-prop-specifier constexpr means that all direct member functions of that class are declared constexpr (9.2.5). *– end note]*

Add after p18 in **[temp.inst]** 13.9.1:

18 ...
[*Example*: The class `S1<T>::Inner1` is ill-formed, no diagnostic required, because it has no valid specializations. `S2` is ill-formed, no diagnostic required, since no substitution into the constraints of its `Inner2` template would result in a valid expression. *– end example]*

19 If a class template is declared with the `constexpr` specifier any implicit instantiation is `constexpr` as well.

Modify **[tab:cpp.predefined.ft]**

`__cpp_constexpr` ~~`201907L`~~**`202002L`**

6 Acknowledgements

Thanks to Ville Voutilainen, Barry Revzin, Matthew Woehlke, Giuseppe D'Angelo, Nevin Liber, Balog Pal, and Joshua Berne for their feedback on the reflector.

Thanks to Jens Maurer for spontaneously jumping on a wording review of this paper.

7 Revision History

Version	Date	Changes
0		Initial draft
1		• Added section about specifier order. • Updated wording.

Bibliography

[P0980R1] Louis Dionne: *"Making std::string constexpr"*, P0980R1, 2019-07-19.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0980r1.pdf

[P1004R2] Louis Dionne: *"Making std::vector constexpr"*, P1004R2, 2019-07-19.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1004r2.pdf

[P1645R1] Ben Deane: *"constexpr for <numeric> algorithms"*, P1645R1, 2019-05-14.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1645r1.html

[P2273R0] Andreas Fertig: *"Making std::unique_ptr constexpr"*, P2273R0, 2020-11-27.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2273r0.pdf

[P1235R0] Bryce Adelstein Lelbach, Hana Dusíková: *"Implicit constexpr"*, P1235R0, 2018.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1235r0.pdf

[P0707R4] Herb Sutter: *"Metaclassfunctions: Generative C++"*, P0707R4, 2019.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0707r4.pdf

[N4885] Thomas Köppe: *"Working Draft, Standard for Programming Language C++"*, N4885, 2021-03-17.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/n4885.pdf

[MSVCVector] MSVC STL: *"P0980R1 constexpr std::string* (#1502)".
https://github.com/microsoft/STL/blob/62137922ab168f8e23ec1a95c946821e24bde230/stl/inc/vector

[GHUPImpl] Andreas Fertig: *"LLVM/Clang constexpr class* implementation on GitHub".
https://github.com/andreasfertig/llvm-project/tree/P2350

[ml16332] Ville Voutilainen: "on *constexpr class* EWG mailing list".
https://lists.isocpp.org/ext/2021/04/16332.php

[MCSrc] Jean-Michaël Celerier: "constexpr class with metaclasses".
https://cppx.godbolt.org/z/oGP5MYcja

[MCSrcTweet] Jean-Michaël Celerier: "constexpr class with metaclasses".
https://twitter.com/jcelerie/status/1380271683408396288