`constexpr class`

Document Number:	**P2350 R2**
Date:	2021-10-14
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 decl-specifiers.

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`:

2 Motivation

Marking all member functions as `constexpr` in a class is an absolutely unnecessary burden for the author of a class. What we can do with `constexpr` has improved over the last couple of years so much that `constexpr` code doesn't differ from the one, which is pure run-time code, a few exceptions aside.

Having to read a class where `constexpr` is attached to each member function is a burden for users of such a class. It disturbs readability. As we still and ever will, write more run-time than compile-time code, the question of whether a certain member function is `constexpr` and by that usable in a compile-time context comes up way less than the other way around. Then if *all* member functions of a class are `constexpr` have that information at the very top at the class definition spares looking up each member function to figure out whether it is usable at compile-time or not.

This paper proposes to introduce a way of marking all member functions in a class as `constexpr` with a single specifier in the class-head, like we can already do it with `final`.

Currently

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

With proposal

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

Currently

```cpp
struct BaseA {
  constexpr bool fun() { return true; }
};

struct DerivedA : BaseA {
  bool run() { return true; }
};

static_assert(DerivedA{}.fun());
//static_assert(DerivedA{}.run());

struct BaseB {
  bool fun() { return true; }
};

struct DerivedB : BaseB {
 constexpr bool run() { return true; }
};

//static_assert(DerivedB{}.fun());
static_assert(DerivedB{}.run());


struct BaseC {
  BaseC() = default;
  BaseC(int) {}
};

struct DerivedC : BaseC {
 constexpr DerivedC(double) : BaseC{} {}

 using BaseC::BaseC;
};

// BaseC::BaseC(int) isn't constexpr
//constexpr DerivedC c1{3};
// DerivedC(double) and BaseC() are constexpr
constexpr DerivedC c2{3.4};


struct BaseD {
  constexpr BaseD(int) {}
};

struct DerivedD : BaseD {
  // Will never produce a
  // constant expression because
  // DerivedD(double) isn't constexpr
  DerivedD(double) : BaseD{2} {}

  using BaseD::BaseD;
};

// BaseD::BaseD(int) is constexpr
constexpr DerivedD d1{3};
// DerivedD(double) isn't constexpr
//constexpr DerivedD d2{3.4};
```

With proposal

```cpp
struct BaseA constexpr {
  bool fun() { return true; }
};

struct DerivedA : BaseA {
  bool run() { return true; }
};

static_assert(DerivedA{}.fun());
//static_assert(DerivedA.run());

struct BaseB {
  bool fun() { return true; }
};

struct DerivedB constexpr : BaseB {
 bool run() { return true; }
};

//static_assert(DerivedB{}.fun());
static_assert(DerivedB{}.run());


struct BaseC {
  BaseC() = default;
  BaseC(int) {}
};

struct DerivedC constexpr : BaseC {
 DerivedC(double) : BaseC{} {}

 using BaseC::BaseC;
};

// BaseC::BaseC(int) isn't constexpr
//constexpr DerivedC c1{3};
// DerivedC(double) and Base() are constexpr
constexpr DerivedC c2{3.4};


struct BaseD constexpr {
  BaseD(int) {}
};

struct DerivedD : BaseD {
  // Will never produce a
  // constant expression because
  // DerivedD(double) isn't constexpr
  DerivedD(double) : BaseD{2} {}

  using BaseD::BaseD;
};

// BaseD::BaseD(int) is constexpr
constexpr DerivedD d1{3};
// DerivedD(double) isn't constexpr
//constexpr DerivedD d2{3.4};
```

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.~~

I received feedback that a `constexpr(false)` could, in fact, be desirable. Aside from the STL, other libraries come with dependencies to 3rd party functions, often C, making it impossible to have the using function to be `constexpr`. See [QTPoint] for an example. `QPoint` is fully `constexpr` except for `toCGPoint`.

Sometimes only a few functions of a class are deliberately implemented out of line in a cpp-file to avoid code bloat. An example is `QRect::operator|(const QRect &r)`[QRect].

[P2448R0] solves this without requireing some `constexpr(false)` fasiclities.

My anser remains no, we don't need `constexpr(false)`.

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 unaffected by 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 static data members?

By this proposal, static data members get implicitly marked constexpr in a constexpr class.

3.7 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.

3.8 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 others don't.

3.9 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, which we have done in [P1004R2] to `std::vector`. This was not considered a breaking change nor an ABI change.

3.10 What about `consteval`

This paper does not propose a class-level `consteval`. The following sections is kept for futur explorations.

For consistency reasons, `consteval` should be allowed like `constexpr`. That being said, `consteval` comes with some things to consider.

Only one of the two keywords should then be allowed in the class-head.

The mental model of a `consteval` class would be that such a class is absolutely compile-time only.

3.10.1 A `consteval` copy constructor

Consider the following example:

```
struct Test consteval {
  bool fun() const { return true; }
};

struct Derived : Test {};

static_assert(Test{}.fun());

consteval auto X()
{
    Test t{};
    Test t2 = t;

    return true;
}

static_assert(X()); // OK

consteval auto D()
{
    Derived t{};
    Derived t2 = t;  // Derived copy ctor must be consteval!

    return true;
}
```

```

static_assert(D());  // ERROR: Test::Test is not a constant expression

namespace working {
    struct Derived : Test {
        Derived() = default;
        consteval Derived(const Derived&) {};  // notice: consteval now!
    };

    consteval auto D()
    {
        Derived t{};
        Derived t2 = t;

        return true;
    }

    static_assert(D());  // OK
}  // namespace working
```

The function `::D` doesn't pass the `static_assert` because it is not a constant expression. The reason is that the copy constructor of `Derived` isn't `constexpr` or `consteval`. Once I add either of these keywords to `D`'s copy constructor, as in `working::Derived`, the code compiles fine. The reason here seems to be the unknown reference of `Derived` when it gets passed to the base classes copy constructor. Barry Rezin has a paper [P2280R0] that aims to relax things here.

3.10.2 Implicit `consteval static` data members

With static data members being implicitly `constexpr` in a `constexpr class` what should we do with static data members in a `consteval class`?

The code on the left compiles even without being a `constexpr class`. However, the standard does not allow `consteval` variables, making the code on the right fail to compile.

```
class Test {
public:
  constexpr static inline int i{}; // OK
};

static_assert(Test::i == 0);
```

```
class Test {
public:
  consteval static inline int i{}; // ERR
};

static_assert(Test::i == 0);
```

3.10.3 Upgrading and Downgrading

Assume a class is marked `constexpr`. 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`?

Options:

A No up- or downgrading is allowed. A `constexpr` class has only `constexpr` member functions, as a `consteval` class has only `consteval` member functions.

B Allow up- or downgrading. A `constexpr` class can have `consteval` member functions, as a `consteval` class can have `constexpr` member functions.

C A `constexpr` class can have `consteval` member functions but not the other way around.

D A `consteval` class can have `constexpr` member functions but not the other way around.

Option D carries the issue that someone can add a `virtual constexpr` function and by that require a `consteval` class to emit a virtual function table that conflicts with the idea of a compile-time-only class.

C seems the best option.

3.11 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.

3.12 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**.

3.13 Order of the specifiers

This paper allows a flexible order of the specifiers `final` and `constexpr` or `consteval`.

```
class Awesome constexpr final {};
class Awesome final constexpr {};
class Awesome consteval final {};
class Awesome final consteval {};
```

4 Controversial parts

4.0.1 The meaning should be that a `constexpr class` type is usable as NTTP

On the reflector, the BSI provided feedback that the NB would support the paper only if `constexpr class` would enforce the type to be usable as an NTTP. As pointed out during the reflector discussion, the meaning would be very weak, to say at least. Consider Listing 4.1. Suppose `MyType` would be `constexpr` by this paper and enforcing the type being usable as an NTTP. Using it with `int` would work as NTTP to `Fun`. However, once it is be used with `A`, which isn't a literal type, it would result in an error.

```
template<typename T>
struct MyType /* constexpr */ {
  T t;
};

template<auto N>
void Fun() {}

struct A {
  virtual ~A() {}
};

int main()
{
  Fun<MyType<int>{}>(); // OK int is a literal type

  Fun<MyType<A>{}>();   // ERROR: A is not a literal type
}
```

Listing 4.1: Don't make a promise you can't keep.

We already get this error from compilers today. The promise to users that `MyType` is always usable as an NTTP is wrong, and by that `constexpr class` meaningless. It would be especially bad to introduce such a behavior as we have a paper [P2448R0] which I support as the right direction, aiming to relax `constexpr` restrictions as long as something is not used in a `constexpr` context.

This paper does *not* enforce the requirement that `constexpr class` means that a type is always usable as a NTTP.

4.0.2 `constexpr class` should ensure constexprness of the inheritance hierarchy

Another point that was mentioned during the reflector discussion was that `constexpr class` should enforce that all base classes are `constexpr` as well.

```
struct Base {
  bool fun() { return true; }
};

struct Derived constexpr : Base { // ERROR: Base is not constexpr
  bool run() { return true; }
};
```

Listing 4.2: Deriving from a non-constexpr class

The first question that comes up is, what if all member functions in `Base` are `constexpr`? Should it then be counted as a `constexpr class`, or is it only a `constexpr class` if `constexpr` is used in the class-head?

Regardless of the answer to the question above, enforcing the base class to be `constexpr` as well limits the design space. `Derived::run` is `constexpr` in Listing 4.2. The constructor of `B` is implicitly `constexpr`. Deriving from a class that isn't entirely `constexpr` and use only the `constexpr` part during constant evaluation is desirable.

This paper does *not* enforce that the entire inheritance hierarchy must be `constexpr` if `constexpr class` is used with a derived class. This is also outlined in 3.7.

4.0.3 The position of the decl-specifier

The position where the `constexpr` should go to carry its meaning as best is already discussed in 3.12. All of these choices leave room for misinterpretation.

The introduction of group member specifiers was mentioned. Such a facility was proposed as [N3955]. As pointed out in 5 that proposal did not reach consensus and another poll during the discussion of P2350 for such a syntax did also not reach consensus.

Down this road, Erich Keane came up with the idea of something like a `constexpr` scope:

```
struct Base {
  constexpr {
    // Every function in here is constexpr.
  }
};
```

However, this approach was not explored by this proposal.

5 Other proposals

During a discussion on the reflector [N3955] was mentioned. It proposes to have group member specifiers allowing things like:

```
class A {
public constexpr:
    // everyhting in here is implicitly constexpr
};
```

This proposal did not reach consensus back in 2014 and a vote during presenting this paper for an access specifier syntax did again not reach consensus.

5.1 What other languages do

D, for example, doesn't require (or have) a specifier like `constexpr` [DCTFE]. It uses Compile Time Function Execution (CTFE) which automatically happens in various contexts.

6 Other parts of the language

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

6.1 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 6.1: 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.

6.2 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.

6.3 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.

6.4 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.

7 Implementation

This proposal was implemented in a fork of LLVM/Clang from the author [GHUPImpl], including `consteval` and `static` data members of a `constexpr class`. The change was small and easy to apply.

8 Polls

8.1 EWG 2021 October 13 (virtual)

Poll: *Having seen what class-level consteval looks like, we still want it in P2350.*

SF	F	N	A	SA
0	1	7	8	0

Result: *Consensus against, we don't want consteval in P2350.*

Poll: *Send P2350 to electronic polling, targeting CWG for C++23.*

SF	F	N	A	SA
4	8	2	3	0

Result: *Consensus*

Poll: *Given that Committee time is limited, we'd like to see a different paper which further explores class-level consteval.*

SF	F	N	A	SA
3	5	8	0	0

Result: *Consensus*

Poll: *Given that Committee time is limited, we'd like to see a different paper which proposes implicit constexpr.*

SF	F	N	A	SA
7	4	1	4	1

Result: *Consensus*

8.2 EWG 2021 June 21 (virtual)

Poll: *we are interested in pursuing a way to specify that multiple/all class members are constexpr, either as suggested in this paper or through another mechanism, considering that time is limited and there is only so much work we can do*

SF	F	N	A	SA
2	5	2	2	0

Result: *?*

Poll: *support class-level consteval at the same time or before supporting constexpr, to inform the design of class constexpr.*

Result: *Consensus*

SF	F	N	A	SA
2	8	3	0	0

Poll: *use access specifier constexpr instead of `class`-level `constexpr`*

SF	F	N	A	SA
0	1	3	4	5

Result: *no Consensus*

Poll: *`class`-level `constexpr` should also affect `friend` functions*

SF	F	N	A	SA
0	1	2	5	1

Result: *no Consensus*

Poll: *it should be ill-formed to put constexpr on a `class` which can't be entirely `constexpr` (e.g. because of it's base or data members not being `constexpr`)*

SF	F	N	A	SA
0	4	2	4	1

Result: *no Consensus*

Poll: *`static` data members should also be `constexpr` in a `constexpr` class*

SF	F	N	A	SA
3	4	0	1	0

Result: *consensus*

Poll: *allow constexpr and final to appear in either order.*

SF	F	N	A	SA
2	6	4	0	0

Result: *consensus*

---

9 Proposed wording

This wording is based 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 and all direct static data members of that class to be constexpr. 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-virt-specifier*$_{opt}$~~ *class-prop-specifier-seq*$_{opt}$ *base-clause*$_{opt}$
 class-key attribute-specifier-seq$_{opt}$ *base-clause*$_{opt}$

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

class-prop-specifier-seq:
 class-prop-specifier
 class-prop-specifier-seq class-prop-specifier

~~*class-virt-specifier:*~~
class-prop-specifier:
 `constexpr`
 `final`

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

6 If a class is marked with the ~~*class-virt-specifier*~~ *class-prop-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*~~ *class-prop-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 Each *class-prop-specifier* shall appear at most once in a complete *class-prop-specifier-seq*.

8 [*Note*: The *class-prop-specifier* constexpr means that all direct member functions of that class and all direct static data members 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 *class-prop-specifier* `constexpr`, any implicit instantiation is also `constexpr`.

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

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

10 Acknowledgements

Thanks to Ville Voutilainen, Barry Revzin, Matthew Woehlke, Giuseppe D'Angelo, Nevin Liber, Balog Pal, Joshua Berne, Anthony Williams, and Andrew Tomazos for their feedback on the reflector.
Thanks to Jens Maurer for spontaneously jumping on a wording review of this paper's R1 and R2.
Thanks to Daveed Vandevoorde for sharing his ideas about `consteval` class.

Thanks to Giuseppe D'Angelo for his feedback on why `constexpr(false)` could be desirable.

11 Revision History

Version	Date	Changes
0		Initial draft
1		• Added section about specifier order. • Updated wording.
2		• More examples. • Added poll results. • Added 3.6 (static data members). • Added reason for `constexpr(false)`.

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

[N3955] Andrew Tomazos: *"Group Member Specifiers"*, N3955, 2014-02-25.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3955.pdf

[P2280R0] Barry Revzin: *"Using unknown references in constant expressions"*, P2280R0, 2021-01-13.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2280r0.html

[P2448R0] Barry Revzin: *"Relaxing some constexpr restrictions"*, P2448R0, 2021-09-28.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/P2448R0.html

[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

[DCTFE] dlang.org "Compile Time Function Execution (CTFE)".
https://dlang.org/spec/function.html#interpretation

[QTPoint] QPoint implementation".
https://github.com/qt/qtbase/blob/9db7cc79a26ced4997277b5c206ca15949133240/src/corelib/tools/qpoint.h

[QRect] QRect implementation".
https://code.woboq.org/qt5/qtbase/src/corelib/tools/qrect.cpp.html#_ZNK5QRectorERKS_