constexpr consternation

Author: Richard Smith
Contact: Doug Gregor <doug.gregor@gmail.com>
Document number: N3308=11-0078
Date: 2011-09-08

History

When constexpr was first proposed, constexpr forward declarations were outlawed, and constexpr functions and constructors were required to always produce a constant expression for any constant expression arguments. Both of these decisions have been (rightly) changed, but the resulting changes have not been fully propagated throughout the standard.

Problems

A number of problems have been found while implementing the FDIS wording for `constexpr`.

Non-defining `constexpr` function declarations

The introduction of forward-declarations of `constexpr` functions by N2826 introduced the possibility of the instantiation of a `constexpr` template function declaration with no accompanying definition. In such a case, it is unspecified whether the instantiated definition should be `constexpr`. Thus it is impossible to determine whether the following class template instantiates to a literal type:

```
template<typename T> struct S { constexpr S(T); };
```

Additionally, while there are a number of restrictions on the argument types and result types of `constexpr` function definitions, there are no such restrictions on the corresponding declarations.

Whether a declared-but-not-defined function is `constexpr` is usually immaterial, but matters when determining whether a type is a literal type. However, the notion of a literal type itself is unnecessary, and the standard could easily make do without it. It does not capture the notion of “a type which has constant expression values”, and makes understanding, implementing, and using `constexpr` harder. A review of every mention of ‘literal type’ in the standard turns up the following:

Types of `constexpr` variables

`constexpr` variables and static data members are required to be of literal type. This constraint is entirely unnecessary and can be deleted: such variables are required to be initialized with a constant expression at the point of declaration; checking whether the type is literal just produces less useful diagnostics.

Requirements on `constexpr` functions and constructors

Arguments and return values of `constexpr` functions and constructors are required to be of literal type. This constraint is partly unnecessary: compilers are free to diagnose such cases anyway, since such functions are required to be able to produce a constant expression when constant expressions are substituted in. However, compilers would not be *required* to diagnose such cases without this check.

Types of constexpr static data member declarations

The type of a constexpr static data member is required to be a literal type. As for constexpr variables, this constraint is unnecessary, since such variables are required to be initialized with a constant expression at the point of declaration.

Value-dependent constants

[temp.dep.constexpr]p2 identifies constants with literal type initialized with value-dependent expressions as being value-dependent. The set of value-dependent expressions is growing in C++11 as a result of this rule; extending it further to all constants initialized with value-dependent expressions does not seem to present a particular issue.

is_literal_type

Generic programming cannot beneficially make use of std::is_literal_type<>, since it does not guarantee that any particular expression will produce a constant expression. However, such detection does not even require compiler support; SFINAE techniques can be used to determine if particular expressions are constant expressions:

```
template<typename T> constexpr bool swallow(T) { return true; }
template<typename T> integral_constant<bool, swallow(T() + T())>
constexpr_addable_helper(int);
template<typename T> false_type constexpr_addable_helper(...);
template<typename T> using constexpr_addable =
decltype(constexpr_addable_helper<T>(0));

static_assert(constexpr_addable<int>(), "");
struct S {}; S operator+(S, S);
static_assert(!constexpr_addable<S>(), "");
```

istream_iterator

Some of the `istream_iterator<T>` constructors are required to be `constexpr` if T is a literal type. The intention is to allow the existing implementation technique of storing an element of type T inline to continue to work. However, it actually rules out this technique: the default and copy constructors of T need not be marked `constexpr`, and if they are not, the `istream_iterator<T>` constructors could not be instantiated as `constexpr`.

Option 1 - tweak definition of literal type

The definition of 'literal type' could be modified such that instantiations of class templates are literal types if the template from which they are instantiated has any `constexpr` constructors or constructor templates other than move/copy constructors. `g++` implements this resolution (perhaps accidentally).

Option 2 - split constexpr requirements

The requirements on `constexpr` functions and constructors could be split into requirements on declarations and requirements on definitions, and then only the requirements on declarations could be considered when determining whether a `constexpr` function template instantiation is `constexpr`.

Option 3 - remove notion of literal type

The notion of a literal type does not seem a useful addition to the existing notion of a constant expression, and removing it would make C++ simpler and easier to learn, and would not hamstring future developments over compatibility concerns.

Comments

In practice, the only use of the notion of a 'literal type' is to force implementers to diagnose certain ill-formed constructs where otherwise no diagnostic would be required. However, removing it at this very late stage seems impractical, so we provide wording for option 2.

Types can become literal after their definition

Despite the clear intentions of the standard, type B is not a literal type:

```
struct A {}; struct B : A {};
```

This is because its constructor is not implicitly defined until it is odr-used, and until that point it has no `constexpr` constructors. This is hard to satisfactorily fix if option 3 (above) is not chosen: in general, determining whether the constructor would be defined `constexpr` requires going though the motions of defining it (including performing any necessary template instantiations and so on), and such work would be required in many contexts which require a literal type.

Comments

We feel the best way to solve this problem is option 3 above (which removes any way to detect whether the constructor is `constexpr` without odr-using it). If that option is not chosen, implementations could be required to determine whether the implicitly-defined constructor would be `constexpr` when the meaning of the program depends on whether the implicit declaration is `constexpr`, or an implicit default constructor could be declared (and defined) `constexpr` if the class' members and bases all have at least one `constexpr` constructor.

constexpr functions cannot use their parameters

[dcl.constexpr](7.1.5)/3 says: "The definition of a `constexpr` function shall satisfy the following constraints: [...] every constructor call and implicit conversion used in initializing the return value shall be one of those allowed in a constant expression." Hence this is ill-formed:

```
        constexpr int f(int a) { return a; }
```

… since the implicit lvalue-to-rvalue conversion on *a* is not allowed in a constant expression.

Comments

The requirement on constructor calls is dubious too: if a constructor might not be called (depending on the values of the parameters) but is nonetheless used by the returned expression, the function should not be ill-formed.

Undefined constexpr functions can sometimes be called in constant expressions

[expr.const](5.19)/2: Calls to undefined `constexpr` functions are allowed within constant expressions in the definition of `constexpr` functions. Thus this is legal:

```
        template<int N> struct S;
        constexpr int f();
        constexpr int g() { return S<f()>::x; }
```

Such calls should not be core constant expressions.

Implicitly generating ill-formed `constexpr` functions

Many `constexpr` function templates can instantiate to functions which are marked `constexpr` but cannot produce constant expressions. For instance:

```
        struct S {}; bool operator<(S, S);
        template<typename T> constexpr T max(T a, T b) { return (a < b) ? b : a; }
        constexpr S m = max(S(), S()); // ill-formed (no diagnostic required)
```

This is ill-formed because the instantiation `max<S>` is defined as a `constexpr` function, but cannot produce a constant expression.

Likewise in this example:

```
        int f();
        struct A { constexpr A(int = f()); };
        struct B : A {};
        B b;
```

Here, the implicit definition of B::B() is `constexpr`, since it obeys the requirements of *[dcl.constexpr](7.1.5)/4*, but it can never produce a constant expression so is ill-formed (again, no diagnostic is required).

And in this example:

```
        struct A { A(); };
        struct B { constexpr B(int); };
        struct C { A a; using B::B; } c;
```
This is ill-formed (no diagnostic required) because C::C() is `constexpr` (because `constexpr` is an inherited constructor characteristic), but again can never produce a constant expression.

Comments

An implementation is, of course, free to simply not diagnose these cases. But we feel that they should be well-formed! We propose that template instantiations and implicitly-defined members be exempt from this rule. Templates would instead follow the rule that there must exist a set of template arguments and function arguments which would allow them to produce a constant expression.
Further, the wording for inherited constructors and defaulted constructors should be tightened such that the above cases do not produce `constexpr` constructors.

Unions and constexpr

Union constructors can be marked `constexpr`, but are required to initialize all non-static data members. Hence only unions with at most one member can have `constexpr` constructors. Likewise a union-like class is required to initialize all of its variant members.

Comments

Union constructors should be required to initialize exactly one member (if the union has any members). Union-like class constructors should be required to initialize all non-variant members, and one variant member in each non-empty anonymous union. `g++` does not require `constexpr` union constructors to initialize any member.

Deleted constexpr constructors and virtual base classes

`constexpr` constructors cannot generally appear in classes with virtual base classes. However, if they are defined as deleted, then they can be. For instance, here:
```
        struct A {};
        struct B : virtual A {
          constexpr B() = delete; };
```
This creates particular problems for option 2 (above) since it means that ‘no virtual base classes’ is not a requirement of a `constexpr` constructor declaration, but is strange and inconsistent in any case.

Forward declarations of types inside and outside typedefs

This is legal:
```
        constexpr int f() {
          typedef struct S;
          return 0;
        }
```
This is not:
```
        constexpr int f() {
          struct S;
          return 0;
        }
```
This is inconsistent, to say the least. Since implementations are required to support the former, it is not unreasonable to require them to also support the latter. Implementation experience shows that supporting full class definitions in `constexpr` functions is also no additional burden.

In-class initializers and `constexpr` constructors

This is ill-formed:

```
        int f();
        struct S {
          int a = f();
```

```
  constexpr S(int b) : a(b) {}
};
```

because a class with a non-constant in-class initializer cannot have any constexpr constructors, even if they don’t use that initializer. This is ill-formed too:

```
struct S {
  int a = 0, b = a;
  constexpr S(int k) : a(k) {} };
```

because b’s initializer is non-constant (even though it could be used unproblematically within a `constexpr` constructor).

constexpr in for-range-declarations

An over-zealous rewording for the resolution of CWG1204 (added in the FDIS) allowed `constexpr` in for-range-declarations. It is (almost) not possible for a for-range-declaration to ever legally be `constexpr`, since its initializer is of the form `*__begin`, where `__begin` is neither `constexpr` nor `const`.

Proposed wording

The following wording changes target option 2 above; most of these changes would be required for any of the options for handling literal types.

[basic.start.init](3.6.2)/2

Variables with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place.
Constant initialization is performed:
— if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.19) and the reference is bound to an lvalue designating an object with static storage duration or to a temporary (see 12.2);
— if an object with static or thread storage duration is initialized by a constructor call, if the constructor is a `constexpr` constructor, if all constructor arguments are constant expressions (including conversions), and if, after function invocation substitution (7.1.5), every ~~constructor call and full-expression in the *mem-initializers* and in the *brace-or-equal-initializers* for non-static data members~~ data member and base class sub-object is initialized by a constant expression;
— if an object with static or thread storage duration is not initialized by a constructor call and if every full-expression that appears in its initializer is a constant expression.
Rationale: a brace-or-equal-initializer within the class definition should not affect behaviour of a constructor which does not use it.

[basic.types](3.9)/10

A type is a *literal type* if it is:
— a scalar type; or
— a reference type; or
— a class type which is either complete or being defined, and whose complete class type (Clause 9) T ~~that has all of the following properties:~~

> ~~— it has a trivial destructor,~~
> ~~— every constructor call and full-expression in the brace-or-equal-initializers for non-static data members (if any) is a constant expression (5.19),~~
> ~~— it~~ is an aggregate type (8.5.1) or has at least one `constexpr` constructor or constructor template that does not have a parameter of type (possibly cv-qualified) reference to T~~is not a copy or move constructor, and~~
> ~~— it has all non-static data members and base classes of literal types~~; or

— an array of literal type.
Rationale: While a class is being defined, it may not yet have any constexpr constructors, but should still be usable as a constexpr member function argument. The requirements of a trivial destructor and literal base and member types have been moved to [dcl.constexpr](7.1.5)/4 to allow such cases to be diagnosed. The requirement on brace-

or-equal-initializers within the class definition is unnecessary and harmful. Finally, a constexpr constructor S(const S&, int) should not be enough to make S a literal type, especially since if the definition of S::S adds a default argument, it would stop being one.

[expr.const](5.19)/2

A *conditional-expression* is a core constant expression unless it involves one of the following as a potentially evaluated subexpression (3.2), but subexpressions of logical AND (5.14), logical OR (5.15), and conditional (5.16) operations that are not evaluated are not considered [Note: An overloaded operator invokes a function. — end note]:
— `this` (5.1) unless it appears as the *postfix-expression* in a class member access expression, including the result of the implicit transformation in the body of a non-static member function (9.3.1);
— an invocation of a function other than a `constexpr` constructor for a literal class or a `constexpr` function [Note: Overload resolution (13.3) is applied as usual — end note];
— an invocation of an undefined `constexpr` function or an undefined `constexpr` constructor~~ outside the definition of a `constexpr` function or a `constexpr` constructor~~;
— an invocation of a `constexpr` function with arguments that, when substituted by function invocation substitution (7.1.5), do not produce a constant expression; [*Example:*

```
constexpr const int* addr(const int& ir) { return &ir; } // OK
static const int x = 5;
constexpr const int* xp = addr(x); // OK: (const int*)&(const int&)x is an
                                   // address contant expression
constexpr const int* tp = addr(5); // error, initializer for constexpr variable not a
constant
                                   // expression; (const int*)&(const int&)5 is not a
constant
                                   // expression because it takes the address of a
temporary
```

— *end example*]
— an invocation of a `constexpr` constructor with arguments that, when substituted by function invocation substitution (7.1.5), do not produce all constant expressions for the initializers of all base class sub-objects and data members~~constructor calls and full-expressions in the *mem-initializers*~~; *[Example:*

```
        int x; // not constant
        struct A {
        constexpr A(bool b) : m(b?42:x) { }
        int m; };
        constexpr int v = A(true).m;  // OK: constructor call initializes
                                      // m with the value 42 after substitution
        constexpr int w = A(false).m; // error: initializer for m is
                                      // x, which is non-constant
```

— *end example]*
— an invocation of a `constexpr` function or a `constexpr` constructor that would exceed the implementation-defined recursion limits (see Annex B);
— a result that is not mathematically defined or not in the range of representable values for its type;
— a lambda-expression (5.1.2);
— an lvalue-to-rvalue conversion (4.1) unless it is applied to
    — a glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression, or
    — a glvalue of literal type that refers to a non-volatile object defined with `constexpr`, or that refers to a sub-object of such an object, or
    — a glvalue of literal type that refers to a non-volatile temporary object whose lifetime has not ended, initialized with a constant expression;
— an lvalue-to-rvalue conversion (4.1) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;
— an id-expression that refers to a variable or data member of reference type unless the reference has a preceding initialization, initialized with a constant expression;
— a dynamic cast (5.2.7);
— a reinterpret_cast (5.2.10);
— a pseudo-destructor call (5.2.4);
— increment or decrement operations (5.2.6, 5.3.2);
— a typeid expression (5.2.8) whose operand is of a polymorphic class type;
— a new-expression (5.3.4);

— a delete-expression (5.3.5);
— a subtraction (5.7) where both operands are pointers;
— a relational (5.9) or equality (5.10) operator where the result is unspecified;
— an assignment or a compound assignment (5.17); or
— a *throw-expression* (15.1).

[stmt.ranged](6.5.4)/2
In the *decl-specifier-seq* of a *for-range-declaration*, each *decl-specifier* shall be ~~either~~ a *type-specifier* ~~or `constexpr`~~.

[dcl.constexpr](7.1.5)/1
The `constexpr` specifier shall be applied only to the definition of a variable, the declaration of a function or function template, or the declaration of a static data member of a literal type (3.9). If
any declaration of a function or function template has a `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier. [Note: An explicit specialization can differ from the template declaration with respect to the `constexpr` specifier. — end note] [Note: Function parameters cannot be declared `constexpr`. — end note] [Example: …]

[dcl.constexpr](7.1.5)/2
A `constexpr` specifier used in the declaration of a function or function template that is not a constructor declares that ~~function~~entity to be a *constexpr function* or a *constexpr function template*, respectively. Similarly, a `constexpr` specifier used in the declaration of a constructor or constructor template ~~declaration~~ declares that ~~constructor~~entity to be a *constexpr constructor* or a *constexpr constructor template*, respectively. `constexpr` functions and `constexpr` constructors are implicitly `inline` (7.1.2).
Rationale: The standard uses these terms, so should define them.

[dcl.constexpr](7.1.5)/3
The ~~definition~~declaration of a `constexpr` function shall satisfy the following ~~constraints~~requirements:
— it shall not be virtual (10.3);
— its return type shall be an incomplete class type or a literal type; and
— each of its parameter types shall be an incomplete class type or a literal type~~;~~.
The definition of a constexpr function shall satisfy the following requirements:
— its function-body shall be = delete, = default, or a *compound-statement* that contains only
  — null statements,
  — static_assert-declarations
  — simple-~~typedef~~ declarations and alias-declarations that do not define objects, references, classes or enumerations,
  — using-declarations,
  — using-directives,
  — and exactly one return statement;
~~— every constructor call and implicit conversion used in initializing the return value (6.6.3, 8.5) shall be one of those allowed in a constant expression (5.19).~~
[Example:
```
constexpr int square(int x)
{ return x * x; }                // OK
constexpr long long_max()
{ return 2147483647; }           // OK
constexpr int abs(int x)
{ return x < 0 ? -x : x; }       // OK
constexpr void f(int x)          // error: return type is void
{ /* ... */ }
~~constexpr int prev(int x)~~
~~{ return --x; }                // error: use of decrement~~
constexpr int g(int x, int n) { ~~// error: body not just “return expr”~~
int r = 1;                       // error: cannot define objects
while (--n > 0) r *= x;          // error: while-statement not permitted
return r; }
```
— end example]
Rationale: The constraints have been split, and renamed to requirements since that is how they are described elsewhere in the standard. Forward-declared functions should be able to have parameters of forward-declared types. Since a function definition requires complete argument and return types, constexpr function definitions still require

their argument and return types to be literal. The requirement on constructor calls and implicit conversions rejects too much, and is unnecessary since such a function is required to be able to return a constant expression.

[dcl.constexpr](7.1.5)/4

~~In a definition~~A declaration of a `constexpr` constructor~~,~~ shall satisfy the following requirements:
— each of the parameter types shall be an incomplete class type or a literal type;
— the class shall not have any virtual base classes;
— each non-static data member and base class shall be of literal type; and
— the complete class type shall have a trivial destructor.

In a definition of a `constexpr` constructor~~addition,~~ either its function-body shall be `= delete` or `= default` or it shall satisfy the following constraints:

~~— the class shall not have any virtual base classes;~~
— its function-body shall not be a function-try-block;
— the compound-statement of its function-body shall contain only
  — null statements,
  — static_assert-declarations
  — simple-~~typedef~~ declarations and alias-declarations that do not define objects, references, classes or enumerations,
  — using-declarations,
  — and using-directives;
— every non-static data member and base class sub-object shall be initialized (12.6.2);
— every constructor involved in initializing non-static data members and base class sub-objects shall be a `constexpr` constructor;
~~— every assignment-expression that is an initializer-clause appearing directly or indirectly within a brace-or-equal-initializer for a non-static data member that is not named by a mem-initializer-id shall be a constant expression; and~~
~~— every implicit conversion used in converting a constructor argument to the corresponding parameter type and converting a full-expression to the corresponding member type shall be one of those allowed in a constant expression.~~

[Example:
```
        struct Length {
        explicit constexpr Length(int i = 0) : val(i) { }
        private:
        int val; };
```
— end example]

Rationale: Same as /3. Some bullets have been moved here from the definition of a literal type, to require such constexpr constructors to be diagnosed. (These diagnostics were permitted anyway, since such a constructor could not produce a constant expression.) The bullet on brace-or-equal-initializers for non-static data members rejects reasonable programs, and the previous bullet already covers the corrected requirements.

[dcl.constexpr](7.1.5)/5

[...] For a ~~`constexpr`~~ function or function template declared with the `constexpr` specifier, if no combination of function argument values and template arguments exist such that the function invocation substitution (preceded by instantiation of any surrounding templates) would produce a constant expression (5.19), the program is ill-formed; no diagnostic required. ~~For a constexpr constructor, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the mem-initializers would be a constant expression (including conversions), the program is ill-formed; no diagnostic required.~~ [*Example: …*] *[Example:*
```
        template<typename T> struct S {
         constexpr int f() { return T().n; }
        };
        struct T { T(); int n; };
        T t = S<T>().f();     // ok

        template<typename T> constexpr T max(T a, T b) { return a < b ? b : a; }
        struct U {}; bool operator<(U, U);
        U u = max(U(), U()); // ok; max<U> is constexpr
```
— end example]

Rationale: constexpr functions should be rejected if that use of the constexpr specifier cannot produce a constant expression. [expr.const] defines what it means for a call to a constructor to be a constant expression; special handling of constexpr constructors is therefore unnecessary.

[dcl.constexpr](7.1.5)/6

If the instantiated template specialization of a constexpr function template or member function declared within~~of~~ a ~~class~~ template ~~would fail to~~ satisf~~y~~ies the requirements for a constexpr function declaration or constexpr constructor declaration, that specialization is ~~not~~ a constexpr function or constexpr constructor. [Note: Otherwise, i~~I~~f the function is a member function it will still be const as described below. — end note] If no specialization of the template would yield a constexpr function or constexpr constructor, the program is ill-formed; no diagnostic required.
Rationale: For consistency, only the declaration of a constexpr function template is analyzed to determine whether an instantiation is constexpr. The sense has been reversed since there is no other normative text which would make constexpr function template instantiations produce constexpr functions. Finally, the wording is extended to handle constexpr member functions in local classes defined within function or class templates.

[class.static.data](9.4.2)/3
If a non-volatile const static data member is of integral or enumeration type, its declaration in the class definition can specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression (5.19). A static data member of literal type can be declared in the class definition with the constexpr specifier; if so, its declaration shall specify a brace-or-equal-initializer in which every constructor call and initializer-clause that is an assignment-expression is a constant expression. [Note: In both these cases, the member may appear in constant expressions. — end note] The member shall still be defined in a namespace scope if it is odr-used (3.2) in the program and the namespace scope definition shall not contain an initializer.

[class.ctor](12.1)/5
A *default constructor* for a class X is a constructor of class X that can be called without an argument. If there is no user-declared constructor for class X, a constructor having no parameters is implicitly declared as defaulted (8.4). An implicitly-declared default constructor is an inline public member of its class, and is constexpr if its implicit definition would be. A defaulted default constructor for class X is defined as deleted if:
— X is a union-like class that has a variant member with a non-trivial default constructor,
— any non-static data member with no brace-or-equal-initializer is of reference type,
— any non-variant non-static data member of const-qualified type (or array thereof) with no brace-or-equal-initializer does not have a user-provided default constructor,
— X is a union and all of its variant members are of const-qualified type (or array thereof),
— X is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),
— any direct or virtual base class, or non-static data member with no brace-or-equal-initializer, has class type M (or array thereof) and either M has no default constructor or overload resolution (13.3) as applied to M's default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or
— any direct or virtual base class or non-static data member has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.
A default constructor is trivial if it is not user-provided and if:
— its class has no virtual functions (10.3) and no virtual base classes (10.1), and
— no non-static data member of its class has a brace-or-equal-initializer, and
— all the direct base classes of its class have trivial default constructors, and
— for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.
Otherwise, the default constructor is non-trivial.

[class.ctor](12.1)/6
A default constructor that is defaulted and not defined as deleted is implicitly defined when it is odr-used (3.2) to create an object of its class type (1.8) or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no ctor-initializer (12.6.2) and an empty compound-statement. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor definition (7.1.5), the implicitly-defined default constructor is constexpr. Before the defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note*: An implicitly-declared default constructor has an exception-specification (15.4). An explicitly-defaulted definition might have an implicit exception-specification, see 8.4. — *end note*]

[class.copy](12.8)/13

A copy/move constructor that is defaulted and not defined as deleted is implicitly defined if it is odr-used (3.2) to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type or when it is explicitly defaulted after its first declaration. [*Note*: The copy/move constructor is implicitly defined even if the implementation elided its odr-use (3.2, 12.2). — *end note*] If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor definition (7.1.5), the implicitly-~~defined~~declared constructor is constexpr.

[class.inhctor](12.9)/2

The constructor characteristics of a constructor or constructor template are
— the template parameter list (14.1), if any,
— the parameter-type-list (8.3.5),
— the exception-specification (15.4), and
— absence or presence of explicit (12.3.1)~~, and~~
~~— absence or presence of constexpr (7.1.5)~~.

[class.inhctor](12.9) Add new paragraph after p9:

An inherited constructor is declared as `constexpr` if its implicit definition would satisfy the requirements of a `constexpr` constructor.

[istream.iterator.cons](24.6.1.1)/1

Effects: Constructs the end-of-stream iterator. If `T()` is a constant expression~~literal type~~, then this constructor shall be a `constexpr` constructor.
Rationale: A literal type need not have a `constexpr` default constructor.

[istream.iterator.cons](24.6.1.1)/5

Effects: Constructs a copy of `x`. If `T(T())` is a constant expression~~literal type~~, then this constructor shall be a `constexpr`~~trivial copy~~ constructor.
Rationale: A literal type need not be trivially-copyable.