Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Compile-Time Invocable Functions

e-invocable-functions

Only functions decorated with **constexpr** can be invoked as part of a **constant expression**.

Description

description

A constant expression is an expression whose value can be determined at compile time — i.e., one that could be used, say, to define the size of a C-style array or as the argument to a **static_assert**:

```
enum { e_SIZE = 5 };             // e_SIZE is a constant expression of value 5.
int a[e_SIZE];                   // e_SIZE must be a constant expression.
static_assert(e_SIZE == 5, "");  //    "     "   " "     "          "
```

Prior to C++11, evaluating a conventional function at compile time as part of a constant expression was not possible:

```
inline const int z() { return 5; }  // OK, returns a nonconstant expression
int a[z()];                         // Error, z() is not a constant expression
static_assert(z() == 5, "");        // Error,  "     "  "  "      "         "
int a[0 ? z() : 9];                 // Error,  "     "  "  "      "         "
```

Developers, in need of such functionality, would use other means, such as template metaprogramming, external code generators, preprocessor macros, or hardcoded constants (as shown above), to work around this deficiency.

As an example, consider a **metaprogram** to calculate the *n*th factorial number:

```
template <int N>
struct Factorial { enum { value = N * Factorial<N-1>::value }; };  // recursive

template <>
struct Factorial<0> { enum { value = 1 }; };                       // base case
```

Evaluating the `Factorial` metafunction above on a constant expression results in a constant expression:

```
static_assert(Factorial<5>::value == 120, "");  // OK, it's a constant expr.
int a[Factorial<5>::value];                     // OK, array of 120 ints
```

Note, however, that the metafunction can be used only with arguments that are themselves constant expressions:

```
const int factorial(const int n)  // returning const int same as just int
{
    static_assert(n >= 0);        // Error, n is not a constant expression.
    return Factorial<n>::value;   // Error,  "   "  "  "     "         "
}
```

Employing this cumbersome work-around leads to code that is difficult both to write and to read and is also non-trivial to compile, often resulting in long compile times. What's more, a separate implementation will be needed for inputs whose values are not compile-time constants.

C++11 introduces a new keyword, **constexpr**, that gives users some sorely needed control over compile-time evaluation. Prepending the declaration[1] of a function with the **constexpr** keyword informs both the compiler and prospective users that the function is eligible for compile-time evaluation and, under the right circumstances, can and will be evaluated *at compile time* to determine the value of a *constant expression*:

```
constexpr int factorial(int n)  // can be evaluated in a constant expression
{
    return n == 0 ? 1 : n * factorial(n - 1);  // single return statement
}
```

In C++11, the body of a **constexpr** function is restricted to a single **return** statement, and any other language construct, such as **if** statements, loops, variable declarations, and so on are forbidden; see *Restrictions on* **constexpr** *function bodies (C++11 only)* on page 15. These seemingly harsh limitations[2], although much preferred to the `Factorial` metafunction (in the code example above), might make optimizing a function's runtime performance infeasible; see *Potential Pitfalls — Prematurely committing to* **constexpr** on page 43. As of C++14, however, many of these restrictions were lifted, though still not all runtime tools are available during compile-time evaluation; see Section 2.2."**??**" on page **??**.

Simply declaring a function to be **constexpr** does not automatically mean that the function *will* necessarily be evaluated at compile time. A **constexpr** function is *guaranteed* to be evaluated at compile time *only* when invoked in a context where a constant expression is required — a.k.a. a **constexpr context**[3]. This can include, for example, the value of a nontype template parameter, array bounds, the first argument to a **static_assert**, or the initializer for a **constexpr** variable (see Section 2.1."**??**" on page **??**). If one attempts to invoke a **constexpr** function in a **constexpr** context with an argument that is not a constant expression, the compiler will report an error:

```
#include <cassert>   // standard C assert macro
#include <iostream>  // std::cout

void f(int n)
{
    assert(factorial(5) == 120);
        // OK, factorial(5) might be evaluated at compile time since 5 is a
        // constant expression but factorial is not used in a
        // constexpr context.

    static_assert(factorial(5) == 120, "");
```

[1] Note that semantic validation of **constexpr** functions occurs only at the point of *definition*. It is therefore possible to *declare* a member or free function to be **constexpr** for which there can be no valid *definition* — e.g., **constexpr void** `f();` — as the return type of a **constexpr** function's *definition* must satisfy certain requirements, including (in C++11 only) that its return type must not be **void**; see *Restrictions on* **constexpr** *function bodies (C++11 only)* on page 15.

[2] At the time **constexpr** was added to the language, it was a feature under development (and still is); see Section 2.2."**??**" on page **??**.

[3] C++20 formalized this notion with the term **manifestly constant evaluated** to capture all places where the value of an expression must be determined at compile time. This new term coalesces descriptions in several places in the Standard where this concept had previously been used without being given a common name.

```
        // OK, guaranteed to be evaluated at compile time since factorial is
        // used in a constexpr context

    std::cout << factorial(n);
        // OK, likely to be evaluated at run time since n is not a constant
        // expression

    static_assert(factorial(n) > 0, "");
        // Error, n is not a constant expression.
}
```

As illustrated above, simply invoking a **constexpr** function with arguments that are constant expressions does *not* guarantee that the function will be evaluated at compile time. The only way to *guarantee* compile-time evaluation of a **constexpr** function is to invoke it in places where a constant expression is mandatory, such as array bounds, static assertions (see Section 1.1."**??**" on page **??**), **alignas** specifiers (see Section 2.1."**??**" on page **??**), nontype template arguments, and so on.

It is important to understand that if the compiler is required to evaluate a **constexpr** function in a constant expression with *compile-time constant* argument values for which the evaluation would require any operation not available at compile time (e.g., **throw**), the compiler will have no choice but to report an error:

```
constexpr int h(int x) { return x < 5 ? x : throw x; }  // OK, constexpr func

int a4[h(4)];  // OK, creates an array of four integers
int a6[h(6)];  // Error, unable to evaluate h on 6 at compile time
```

In the code snippet above, although we are able to size the *file-scope*[4] `a4` array because the path of execution within the valid **constexpr** function `h` does not involve a **throw**, such is not the case with `a6`. That a valid **constexpr** function can be invoked with compile-time constant arguments and still not be evaluable at compile time is noteworthy.

So far we have discussed **constexpr** functions in terms of *free* functions. As we shall see, **constexpr** can also be applied to *free*-function *templates*, *member* functions (importantly, constructors), and *member*-function templates; see **constexpr** *member functions* on page 13. Just as with free functions, only **constexpr** member functions are eligible to be evaluated at compile time.

What's more, we'll see that there is a category of user-defined types — called **literal types** — whose *operational definition*[5] (for now) is that at least one of its values can

[4] A common extension of popular compilers to allow (by default) variable-length arrays within function bodies but (as illustrated above) *never* at *file* or *namespace* scope:

```
void g()
{
    int a4[h(4)];  // OK, creates an array of four integers
    int a6[h(6)];  // Warning: ISO C++ forbids variable-length array a6.
                   // But with some compilers, h(6) might be invoked at
                   // run time and throw.
}
```

It is only by compiling with `-Wpedantic` that we get even so much as a warning!

[5] By *operational definition* here, we mean a rule of thumb that would typically hold in practice; see

participate in constant expressions:

```
struct Int  // example of a literal type
{
    int d_val;                                      // plain old int data member
    constexpr Int(int val) : d_val(val) { }         // constexpr value constructor
    constexpr int val() const { return d_val; }     // constexpr value accessor
              int dat() const { return d_val; }     // nonconstexpr accessor
};

constexpr int f(){ return Int(5).d_val; }  // OK, constexpr value constructor
constexpr int g(Int i){ return i.val(); }  // OK, constexpr value accessor
constexpr int h(Int i){ return i.dat(); }  // Error, nonconstexpr accessor
```

The basic, intuitive idea of what makes the user-defined `Int` type above a literal type is that it is possible to initialize objects of this type in a **constexpr** context[6]. What makes `Int` a "useful" literal type is that there is at least one way of extracting a value (either directly or via a **constexpr** accessor) such that the programmer can make use of it — typically at compile time. We might, however, imagine a use for a valid literal type that could be constructed at compile time but not otherwise *used* until run time:

```
class StoreForRt   // compile-time constructible (only) literal type
{
    int d_value;   // There is no way of accessing this value at compile time.

public:
    constexpr StoreForRt(int value) : d_value(value) { }  //       constexpr
    int value() const { return d_value; }                  // not constexpr
};
```

Contrived though it might seem, the example code above is representative of an application of **constexpr** where the construction of an object can benefit from compile-time optimization whereas access to the constructed data cannot. It is instructive, however, to first prove that such an object can in fact be constructed at compile time without employing other C++11 features. To that end, we will create a wrapper literal type, `W`, that contains both a member object of type `StoreForRt` and also one of type **int**. For `W` to be a literal type, both of its members must themselves be of literal type, and **int**, being a built-in type (all of which are literal types), is one:

```
struct W  // used to demonstrate compile-time constructability of StoreForRt
{
    StoreForRt d_i;                                 // usable only at run time
    int        d_j;                                 // usable at compile time
    constexpr W(int i, int j) : d_i(i), d_j(j) { }  // constexpr constructor
};
```

Literal types (defined) on page 25.

[6] This initialization can be made possible through a **constexpr** constructor or through a type that can be **list initialized** without needing to invoke any non**constexpr** constructors; see Section 2.1."??" on page ?? and Section 1.2."??" on page ??.

```
static_assert(W(1,2).d_j == 2, "");
    // OK, can use W in a constexpr context so StoreForRt is of literal type

static_assert(StoreForRt(5).value() == 5, "");  // Error, value not constexpr
    // There is no way we can access d_value at compile time.
```

As the example code above demonstrates, `StoreForRt` is a literal type because it is used to declare a data member of a user-defined type, `W`, that in turn has been demonstrated to be *used* in a context that requires a constant expression. It is not, however, possible to do anything more with that constructed object at compile time (except for obtaining certain generic compile-time properties, such as its size (**sizeof**) or alignment; see Section 2.1."**??**" on page **??**).

As it happens, the compiler doesn't actually care whether it can extract values from the compile-time-constructed object's data members: The compiler cares only that it can do — at compile time — all the evaluations demanded of it, with the assumption that those will involve a minimum of creating and destroying such objects; see *Literal types (defined)* on page 25.

To demonstrate that the same object can be (1) *constructed* at compile time and (2) *used* at run time, we will need to resort to use of a C++11 companion feature of **constexpr** functions, namely **constexpr** variables (see Section 2.1."**??**" on page **??**):

```
constexpr StoreForRt x(5);  // OK, object x constructed in a constexpr context

int main() { return x.value(); }  // OK, x.value() used (only) at run time
```

Only literal types are permitted as *parameters* and return types for **constexpr** functions:

```
constexpr int  f11(StoreForRt x) { return 0; }  // OK, x is a literal type
constexpr void f14(StoreForRt x) { }            // OK, in C++14
```

See **constexpr**-*function parameter and return types* on page 24.

constexpr is part of the public interface

-the-public-interface

When a **constexpr** function is invoked with an argument that is *not* known at compile time, compile-time evaluation of the function itself is not possible, and that invocation simply cannot be used in a context where a compile-time constant is required; runtime evaluation, however, is still permitted:

```
      int  i = 10;  // modifiable int variable
const int  j = 10;  // unmodifiable int variable (implicitly constexpr)
      bool mb = 0;  // modifiable bool variable

constexpr int f(bool b) { return b ? i : 5; }  // sometimes works as constexpr
constexpr int g(bool b) { return b ? j : 5; }  //    always works as constexpr

static_assert(f(mb), "");  // Error, mb is not usable in a constant expression.
static_assert(f(0),  "");  // OK
static_assert(f(1),  "");  // Error, i is *not* usable in a constant expression.

static_assert(g(mb), "");  // Error, mb is not usable in a constant expression.
```

```
static_assert(g(0),  "");  // OK
static_assert(g(1),  "");  // OK, j *is* usable in a constant expression.

int xf = f(mb);  // OK, runtime evaluation of f
int xg = g(mb);  // OK, runtime evaluation of g
```

In the example above, `f` can sometimes be used as part of constant expression but only if its argument is itself a constant expression and `b` evaluates to **`false`**. Function `g`, on the other hand, requires only that its argument be a constant expression for it to always be usable as part of a constant expression. If there is not at least one set of compile-time constant argument values that would be usable at compile time then it is **ill formed, no diagnostic required (IFNDR)**:

```
constexpr int h1(bool b) { return f(b); }
    // OK, there is a value of b for which h1 can be evaluated at compile time.

constexpr int h2() { return f(1); }
    // There's no way to invoke h2 so that it can be evaluated at compile time.
    // (This function is ill formed, no diagnostic required.)
```

Here `h1` is well formed since it can be evaluated at compile time when the value of `b` is **`true`**; `h2`, on the other hand, is ill formed because it can *never* be evaluated at compile time. A sophisticated analysis would, however, be required to establish such a proof, and popular compilers often do not currently try; future compilers are, of course, free to do so.

Being part of the user interface, a function marked as being **`constexpr`** might suggest (albeit wrongly) to some prospective clients that the function will *necessarily* support compile-time evaluation whenever it is invoked with compile-time constant arguments. Although *adding* a **`constexpr`** specifier to a function between library releases is not a problematic API change, *removing* a **`constexpr`** specifier definitely is, because existing users might be relying on compile-time evaluation in their code. Library developers have to make a conscious decision as to whether to mark a function **`constexpr`** — especially with the heavy restrictions imposed by the C++11 Standard — since improving the implementation of the function while respecting those restrictions might prove insurmountable; see *Potential Pitfalls — Prematurely committing to* **`constexpr`** on page 43.

d-definition-visibility

Inlining and definition visibility

A function that is declared **`constexpr`** is (1) implicitly declared **`inline`** and (2) automatically eligible for compile-time evaluation. Note that adding the **`inline`** specifier to a function that is already declared **`constexpr`** has no effect:

```
       constexpr int f1() { return 0; } // automatically inline
inline constexpr int f1();              // redeclares the same f1() above
```

As with all **`inline`** functions, it is an **one-definition rule (ODR)** violation if definitions in different translation units within a program are not token-for-token the same. If definitions do differ across translation units, the program is ill formed, no diagnostic required (IFNDR):

```
// file1.h
         inline int f2() { return 0; }
```

```
        constexpr int f3() { return 0; }

// file2.h
           inline int f2() { return 1; }  // Error, no diagnostic required
        constexpr int f3() { return 1; }  // Error, no diagnostic required
```

When a function is declared **constexpr**, *every* declaration of that function, including its definition, must also be explicitly declared **constexpr** or else the program is ill formed:

```
constexpr int f4();
constexpr int f4() { return 0; }  // OK, constexpr matching exactly

constexpr int f5() { return 0; }
           int f5() { return 0; }  // Error, constexpr missing

           int f6();
constexpr int f6() { return 0; }  // Error, constexpr added
```

An explicit specialization of a function template declaration may, however, differ with respect to its **constexpr** specifier. For example, a general function template (e.g., `func1` in the code snippet below) might be declared **constexpr** whereas one of its explicit specializations (e.g., `func1<int>`) might not be:

```
template <typename T>    // general function template declaration/definition
constexpr bool func1(T)  // general template is declared constexpr
{
    return true;
}

template <>              // explicit specialization definition
bool func1<int>(int)     // The explicit specialization is not constexpr.
{
    return true;
}

static_assert(func1('a'), "");  // OK, general function template is constexpr.
static_assert(func1(123), "");  // Error, int specialization is not constexpr.
```

Similarly, the roles can be reversed where only an explicit specialization (e.g., `func2<int>` in the example below) is **constexpr**:

```
template <typename T> bool func2(T)        { return true; }  // general template
template <> constexpr bool func2<int>(int) { return true; }  // specialization

static_assert(func2('a'), "");  // Error, general template is not constexpr.
static_assert(func2(123), "");  // Ok, int specialization is constexpr.
```

Just as with any other function, a **constexpr** function may appear in an expression before its body has been seen. A **constexpr** function's definition, however, must appear before that function is evaluated to determine the value of a *constant expression*:

```
constexpr int f7();                 // declared but not yet defined
```

```
constexpr int f8() { return f7(); }  // defined with a call to f7
constexpr int f9();                  // declared but not defined in this TU

int main()
{
    return f8() + f9();  // OK, presumes f7 and f9 *are* defined and linked
                         // with this TU
}

static_assert(0 == f8(), "");  // Error, body of f7 has not yet been seen.
static_assert(0 == f9(), "");  // Error,   "  "  f9  "   "   "   "     "

constexpr int f7() { return 0; }  // definition matching *forward declaration*

static_assert(0 == f8(), "");  // OK, body of f7 is visible from here
static_assert(0 == f9(), "");  // Error, body of f9 has not yet been seen

// Oops, failed to define f9 in this translation unit; compiler *might* warn
```

In the example code above, we have declared three **constexpr** functions: `f7`, `f8`, and `f9`. Of the three, only `f8` is defined ahead of its first use. Any attempt to evaluate a **constexpr** function whose definition has not yet been seen — either directly (e.g., `f9`) or indirectly (e.g., `f7` via `f8`) — in a **constexpr** context results in a compile-time error. Notice that, when used in expressions whose value does not need to be determined at compile time (e.g., the **return** statement in `main`), there is no requirement to have seen the body. The compiler is, of course, still free to optimize, and, depending on the optimization level, it might substitute the function bodies **inline** and perform constant folding to the extent possible. Note that, in this case, `f9` was not defined anywhere within the TU. Just as with any other **inline** function whose definition is never seen, many popular compilers will warn if they see any expressions that might invoke such a function, but it is not ill formed because the definition could (by design) reside in some other TU (see also Section 2.1."**??**" on page **??**.

However, when a **constexpr** function is *evaluated* to determine the value of a constant expression, its body, and anything upon which it depends must have already been seen; notice that we didn't say "appears as part of a constant expression" but instead said "is evaluated to determine the value of a constant expression."

We *can* have something that is not itself a (*compile-time*) constant expression (or even one that is convertible to **bool**) *appear* as a part of a constant expression provided* that it never actually gets evaluated at compile time:

```
static_assert(false ? throw : true,  "");  // OK
static_assert(true  ? throw : true,  "");  // Error, throw not constexpr
static_assert(true  ?  true : throw, "");  // OK
static_assert(false ?  true : throw, "");  // Error, throw not constexpr

static_assert((true, throw), "");  // Error, throw not convertible to bool
static_assert((throw, true), "");  // Error, throw is not constexpr

extern volatile bool x;
static_assert((true, x), "");      // Error, x not constexpr
```

```
static_assert((x, true), "");      // Error,  "   "      "

static_assert(true || x,    "");   // OK
static_assert(x    || true, "");   // Error, x not constexpr
```

Note that the *comma* (`,`) **sequencing operator** incurs evaluation of both of its arguments whereas the *logical-or* (`||`) **operator** requires only that its two arguments be convertible to **`bool`**, where actual evaluation of the second argument might be short circuited; see ."**??**" on page **??**[**AUs: there is no feature called "Implicit Conversion"**].

mutual-recursion

Mutual recursion

Mutually recursive functions can be declared **`constexpr`** so long as neither is called in a **`constexpr`** context until both definitions have been seen:

```
constexpr int gg(int n);       // forward declaration

constexpr int ff(int n)        // declaration and definition
{
    return (n > 0) ? gg(n - 1) + 1 : 0;
}

int hh()
{
    return ff(1) + gg(2);  // OK, not a *constexpr context*
}

static_assert(ff(3), "");      // Error: body of gg has not yet been seen
static_assert(gg(4), "");      // Error:  "   "   "    "   "   "   "    "

constexpr int gg(int n)        // redeclaration and definition
{
    return (n > 0) ? ff(n - 1) + 1 : 0;
    static_assert(ff(5), "");  // Error: body of gg has not yet been seen
    static_assert(gg(6), "");  // Error:  "   "   "    "   "   "   "    "
}
static_assert(ff(7), "");      // OK: bodies of ff and gg have now been seen
static_assert(gg(8), "");      // OK:    "   "   "    "   "   "   "   "    "
```

In the example code above, we have two recursive functions, `ff` and `gg`, with `gg` being forward declared. When used within (non**`constexpr`**) function `hh`, the mutually recursive calls between `ff` and `gg` are not evaluated until the compiler has seen the bodies of both `ff` and `gg`, i.e., at run time. Conversely, the **`static_assert`**s of `ff(3)` and `gg(4)` are **`constexpr`** contexts and are ill formed because the body of `gg` has not yet been seen at that point in the compilation. The **`static_assert`**s within the function body of `gg` are similarly evaluated at the point they are seen during compilation, where `gg` is not yet (fully) defined and callable, and so are also ill formed. Finally, the **`static_assert`**s of `ff(7)` and `gg(8)` *can* be evaluated at compile time because the bodies of both `ff` and `gg` have both been seen by the compiler by that point in the compilation.

m-and-function-pointers

The type system and function pointers

Similarly to the **inline** keyword, marking a function **constexpr** does *not* affect its type; hence, it is not possible to have, say, two overloads of a function that differ only on whether they are **constexpr** or to define a pointer to exclusively **constexpr** functions:

```
constexpr int f(int) { return 0; }  // OK
int f(int)           { return 0; }  // Error, int f(int) is now multiply defined.

typedef constexpr int(*MyFnPtr)(int);
    // Error, constexpr cannot appear in a typedef declaration.

void g(constexpr int(*MyFnPtr)(int));
    // Error, a parameter cannot be declared constexpr.
```

If a function pointer is not itself declared **constexpr**, its value cannot be read as part of evaluating a constant expression. If the function pointer *is* **constexpr** but points to a non**constexpr** function, it *cannot* be used to invoke that function at compile time:

```
constexpr bool g() { return true; }  //    constexpr function returning true
          bool h() { return true; }  // nonconstexpr function returning true

typedef bool (*Fp)();  // pointer to function taking no args. and returning bool

constexpr Fp m = g;  //    constexpr pointer to a    constexpr function
          Fp n = g;  // nonconstexpr pointer to a    constexpr function
constexpr Fp p = h;  //    constexpr pointer to a nonconstexpr function
          Fp q = h;  // nonconstexpr pointer to a nonconstexpr function
constexpr Fp r = 0;  //    constexpr pointer having nullptr (address) value

static_assert(p == &h, "");  // Ok, reading the value of a constexpr pointer
static_assert(q == &h, "");  // Error, q is not a constexpr pointer.
static_assert(r == 0,  "");  // Ok, reading the value of a constexpr pointer
static_assert(m(), "");      // Ok, invoking a constexpr function through a
                             //     constexpr pointer
static_assert(n(), "");      // Error, n is not a constexpr pointer.
static_assert(p(), "");      // Error, can't invoke a nonconstexpr function
static_assert(q(), "");      // Error, q is not a constexpr pointer.
static_assert(r(), "");      // Error, 0 doesn't designate a function.
```

stexpr-member-functions

constexpr member functions

Member functions — including special member functions (see **??** on page **??**[**AUs: there is no section called "special member functions"**]), such as *constructors* (but not *destructors*; see *Literal types (defined)* on page 25) — can be declared to be **constexpr**:

```
class Point1
{
    int d_x, d_y;  // two ordinary int data members
public:
    constexpr Point1(int x, int y) : d_x(x), d_y(y) { }  // OK, *is* constexpr
```

```
    constexpr int x()       { return d_x; }  // OK, *is* constexpr
              int y() const { return d_y; }  // OK, is *not* constexpr
};
```

Simple classes, such as `Point1` (in the code snippet above), having at least one **constexpr** constructor that is not a *copy* or *move* constructor as well as a **constexpr** accessor (or public data member) and satisfying the other requirements of being a literal type — see *Literal types (defined)* on page 25 — can be used as part of constant expressions:

```
int ax[Point1(5, 6).x()];  // Ok, array of 5 ints
int ay[Point1(5, 6).y()];  // Error, accessor y is not declared constexpr.
```

Member functions decorated with **constexpr** are implicitly **const**-qualified in C++11 (but not in C++14 — see Section 2.2."**??**" on page **??**):

```
struct Point2
{
    int d_x, d_y;                                          // same as for Point1
    constexpr Point2(int x, int y) : d_x(x), d_y(y) { }  //   "   "  "     "

    constexpr int& x() { return d_x; }                     // accessor #1
        // Error, binding int& reference to const int discards qualifiers.

    constexpr const int& y() const { return d_y; }         // accessor #2
        // OK, the 2nd const qualifier is redundant (but only in C++11).

    constexpr const int& y() { return d_y; }               // accessor #3
        // Error, redefinition of constexpr const int& Point::y() const
};
```

In the `Point2` **struct** example above, accessor #1 is implicitly declared **const** in C++11 (but not C++14); hence, the attempt to return a modifiable *lvalue* reference to the implicitly **const** `d_x` data member erroneously discards the **const** qualifier. Had we declared the **constexpr** function to return a **const** reference, as we did for accessor #2, the code would have compiled just fine. Note that the explicit **const** member-function qualifier (the second **const**) in accessor #2 is redundant in C++11 (but not in C++14); having it there ensures that the meaning will not change when this code is recompiled under a subsequent version of the language. Lastly, note that omitting the member-function qualifier in accessor #3 fails to produce a distinct overload in C++11 (but would in C++14).

Because declaring a member function to be **constexpr** implicitly makes it **const**-qualified (C++11 only), there can be unintended consequences:

```
struct Point3
{
    int d_x, d_y;                                          // same as for Point1
    constexpr Point3(int x, int y) : d_x(x), d_y(y) { }  //   "   "  "     "

    constexpr int x() const { return d_x; }  // OK
    constexpr int y()       { return d_y; }  // OK, const is implied in C++11.
```

```
            int setX(int x) { return d_x = x; }  // OK, but not constexpr
    constexpr int setY(int y) { return d_y = y; }  // Error, implied const

    constexpr Point3& operator=(const Point3& p);
        // Error, copy (and move) assignment can't be constexpr in C++11.

};
```

Notice that declaring the "setter" member function `setY` (in the code example above) to be **`constexpr`** implicitly qualifies the member function as being **`const`**, thereby making it an error for any **`constexpr`** member function to attempt to modify its own object's data members. The inevitable corollary is that any reasonable implementation of *copy* or *move* assignment cannot be declared **`constexpr`** in C++11 (but can be as of C++14).

Finally, **`constexpr`** member functions cannot be **`virtual`**[7] but can co-exist in the same class with other member functions that *are* virtual.

ion-bodies-(c++11-only)

Restrictions on `constexpr` function bodies (C++11 only)

The list of C++ programming features permitted in the bodies of **`constexpr`** functions for C++11 is small and reflective of the nascent state of this feature when it was first standardized. To begin, the body of a **`constexpr`** function is not permitted to be a **function-try-block**:

```
          int g1()     { return 0; }                // OK
constexpr int g2()     { return 0; }                // Ok, no try block
          int g3() try { return 0; } catch(...) {}  // OK, not constexpr
constexpr int g4() try { return 0; } catch(...) {}  // Error, not allowed
```

C++11 **`constexpr`** functions that are not *deleted* or *defaulted* (see Section 1.1."**??**" on page **??** and Section 1.1."**??**" on page **??**, respectively) may consist of only **null statements**, static assertions (see Section 1.1."**??**" on page **??**), **`using`** **declarations**, **`using`** **directives**, **`typedef`** declarations, and alias declarations (see Section 1.1."**??**" on page **??**) that do not define a class or enumeration. Other than constructors, the body of a **`constexpr`** function must include exactly one **`return`** statement. A **`constexpr`** constructor may have a member-initializer list but no other additional statements (but see *Constraints specific to constructors* on page 16). Use of the **ternary operator**, **comma operator**, and recursion are allowed:

```
constexpr int f(int x)
{
    static_assert(sizeof(int) == 4, "");   // OK, static assertion
    using MyInt = int;                     // OK, type alias
    return x > 5 ? x : f(x + 2), f(x + 1); // OK, ternary, comma, and recursion
    ;; ;   ; { ; { ; ; } }                 // OK, null statements/nested blocks
}
```

Many familiar programming constructs such as runtime assertions, local variables, **`if`** statements, modifications of function parameters, and **`using`** directives that define a type are, however, *not* permitted (in C++11):

[7] C++20 allows **`constexpr`** member functions to be **`virtual`** (?).

```
constexpr int g(int x)
{
    assert(x < 100);          // Error, no runtime asserts
    int y = x;                // Error, no local variables
    if (x > 5) { return x; }  // Error, no if statements
    using S = struct { };     // Error, no aliases that define types
    return x += 3;            // Error, no compound assignment
}
```

The good news is that the aforementioned restrictions on the kinds of constructs that are permitted in **constexpr** function bodies are significantly relaxed as of C++14; see Section 2.2."**??**" on page **??**.

Irrespective of the *kinds* of constructs that are allowed to appear in a **constexpr** function body, every invocation (evaluation) of a function, a constructor, or an implicit conversion operator in the **return** statement must itself be usable in some (at least one) constant expression, which means the corresponding function *must* (at a minimum) be declared **constexpr**:

```
          int ga() { return 0; }  // nonconstexpr function returning 0
constexpr int gb() { return 0; }  //    constexpr function returning 0

struct S1a {           S1a() { } };  // nonconstexpr default constructor
struct S1b { constexpr S1b() { } };  //    constexpr default constructor

struct S2a { operator int() { return 5; } };      // nonconstexpr conversion
struct S2b { constexpr operator int() { return 5; } }; // constexpr conversion

constexpr int f1a() { return ga(); }  // Error, ga is not constexpr.
constexpr int f1b() { return gb(); }  // Ok, gb is constexpr.

constexpr int f2a() { return S1a(), 5; }  // Error, S1a ctor is not constexpr.
constexpr int f2b() { return S1b(), 5; }  // Ok, S1b ctor is constexpr.

constexpr int f3a() { return S2a(); } // Error, S2a conversion is not constexpr.
constexpr int f3b() { return S2b(); } // Ok, S1b conversion is constexpr.
```

Note that non**constexpr** implicit conversions, as illustrated by `f3a` above, can also result from a non**constexpr**, non**explicit** constructor that accepts a single argument.

cific-to-constructors

Constraints specific to constructors

In addition to the general restrictions on a **constexpr** function's body (see **constexpr**-*function parameter and return types* on page 24) and its allowed parameter and return types (see **constexpr**-*function parameter and return types* on page 24), there are several additional requirements specific to constructors.

1. The body of a **constexpr** constructor is restricted in the same way as any other **constexpr** function, with the obvious lack of a **return** statement being allowed. Hence, the body of **constexpr** constructor, in addition to not being permitted within a function **try** block (like any other **constexpr** function) must be essentially empty with just a very few exceptions — e.g., null statements, **static_assert** declarations,

typedef declarations (see also Section 1.1."**??**" on page **??**) that do not define classes or enumerations, **using** declarations, and **using** directives:

```
namespace n           // enclosing namespace
{

class C { /*...*/ };  // arbitrary class definition

struct S
{
    constexpr S(bool) try { } catch (...) { }  // Error, function try block
              S(char) try { } catch (...) { }  // OK, not declared constexpr

    constexpr S(int)
    {
        ;                       // OK, null statement
        {}                      // Error, though accepted by most compilers
        static_assert(1, "");   // OK, static_assert declaration
        typedef int Int;        // OK, simple typedef alias
        using Int = int;        // OK, simple using alias
        typedef enum {} E;      // Error, typedef used to define enum E
        using n::C;             // OK, using declaration
        using namespace n;      // OK, using directive
    }
};

}  // close namespace
```

2. All non**static** data members and base-class subobjects of a class must be initialized by a **constexpr** constructor,[8] and the initializers themselves must be usable in a *constant expression*. Members with a trivial default constructor must be explicitly initialized in the member-initializer list or via a **default member initializer** (i.e., they cannot be left in an uninitialized state):

```
struct B  // constexpr constructible only from argument convertible to int
{
    B() { }
    constexpr B(int) { }  // constexpr constructor taking an int
};

struct C  //  constexpr default constructible
{
    constexpr C() { }  // constexpr default constructor.
};

struct D1 : B  // public derivation
```

[8] The requirement that all members and base classes be initialized by a constructor that is explicitly declared **constexpr** is relaxed in C++20 provided that uninitialized entities are not accessed at compile time.

```
{
    constexpr D1() { }  // Error, B has nonconstexpr *default* constructor
};

struct D2 : B  // public derivation
{
    int d_i;  // nonstatic, trivially constructible data member
    constexpr D2(int i) : B(i) { }  // Error, doesn't initialize d_i
};

          int f1() { return 5; }  // nonconstexpr function
constexpr int f2() { return 5; }  //    constexpr function

struct D3 : C  // public derivation
{
    int d_i = f1();  // initialization using nonconstexpr function
    int d_j = f2();  // initialization using    constexpr function

    constexpr D3() { }  // Error, d_i initialized using nonconstexpr function

    constexpr D3(int i) : d_i(i) { }  // OK, d_i set from initializer list
};
```

The example code above illustrates various ways in which a base class or non**static** data member might fail to be initialized by a constructor that is explicitly declared **constexpr**. In the final derived class, `D3`, we note that there are two data members, `d_i` and `d_j`, having member initializers that use a non**constexpr** function, `f1`, and a **constexpr** function, `f2`, respectively. The implementation of the **constexpr** *default* constructor , `D3()`, is erroneous because data member `d_i` would be initialized by the `non`constexpr function `f1` at run time. On the other hand, the implementation of the *value* constructor, `D3(int)`, is fine because the data member `d_i` is set in the member-initializer list, thereby enabling compile-time evaluation.

3. Defining a constructor to be **constexpr** requires that the class have no **virtual** base classes[9]:

```
struct B { constexpr B();  /*...*/ };  // some arbitrary base class

struct D : virtual B  // Virtual base classes preclude constexpr constructors.
{
    constexpr D(int) { }   // Error, class D has virtual base class B.
};
```

4. Like any **special member function**, a constructor that is *explicitly* declared to be **constexpr** can always be suppressed using **=delete** (see Section 1.1."**??**" on

[9] C++20 removes the restriction that a constructor cannot be **constexpr** if the class has any virtual base classes.

page **??**).[10]. If a constructor is implemented using `=default`, however, an error will result *unless* the defaulted definition would have been implicitly **constexpr** (see Section 1.1."**??**" on page **??**):

```
struct S1
{
    S1() { };           // nonconstexpr *default* constructor
    S1(const S1&) { };  //  "      "      *copy*         "
    S1(char) { };       //  "      "      *value*        "
};

struct S2
{
    S1 d_s1;
    constexpr S2() = default;          // default constructor
        // Error, S1's *default* constructor isn't constexpr.

    constexpr S2(const S2&) = delete;  // copy constructor
        // OK, make declaration *inaccessible* and suppress implementation

    S2(char c) : d_s1(c) { }           // value constructor
       // OK, this constructor is *not* declared to be constexpr.
};
```

In the example above, explicitly declaring the *default* constructor of `S2` to be **constexpr** is an error because an implicitly defined *default* constructor would not have been **constexpr**. Using `=delete` *declares* but does not *define* a **constexpr** function; hence, no semantic validation with respect to **constexpr** is applied to `S2`'s (suppressed) *copy* constructor. Because `S2`'s *value* constructor (from **char**) is not explicitly declared **constexpr**, there is no issue with delegating to its non**constexpr** member *value*-constructor counterpart.

5. An implicitly defined *default* constructor (generated by the compiler) performs the set of initializations of the class that would be performed by a user-written default constructor for that class having no member-initializer list and an empty function body. If such a user-defined *default* constructor would satisfy the requirements of a **constexpr** constructor, the implicitly defined *default* constructor is a **constexpr** constructor (and similarly for the implicitly defined *copy* and *move* constructors) irrespective of whether it is explicitly declared **constexpr**. Explicitly declaring a defaulted constructor **constexpr** that is *not* inherently **constexpr** is, however, a compile-time error (see Section 1.1."**??**" on page **??**):

```
struct I0  { int i; /* implicit default ctor */ };  // OK, I0 is a literal type.

struct I1a { int i;           I1a()       { } };  // OK, i is not initialized.
struct I1b { int i;  constexpr I1b()       { } };  // Error, i not initialized.
```

[10] Deleting a function explicitly declares it, makes that declaration *inaccessible*, and suppresses generation of an implementation; see **??** on page **??**[**AUs: There is no subsection called "Detecting literal types"**]

```
struct I2a { int i;            I2a() = default; };  // OK, but not constexpr
struct I2b { int i;  constexpr I2b() = default; };  // Error, i is not initialized.

struct I3a { int i;            I3a() : i(0) { } };  // OK, i is initialized.
struct I3b { int i;  constexpr I3b() : i(0) { } };  // OK, I3b's a literal type.

struct S0  { I3b i; /* implicit default ctor */ };  // OK, S0 is a literal type.

struct S1a { I3b i;            S1a()        { } };  // OK, i is initialized.
struct S1b { I3b i;  constexpr S1b()        { } };  // OK, S1b's a literal type.

struct S2a { I3b i;            S2a() = default; };  // OK, S2a's a literal type.
struct S2b { I3b i;  constexpr S2b() = default; };  // OK, S2b's a literal type.
```

The example code above attempts to illustrate the subtle differences between a data member of *scalar* literal type** (e.g., **int**) and one of *user-defined* literal type** (e.g., `I3b`). The first difference, illustrated by `I1a` versus `S1a`, is that the former always leaves its own data member, `i`, uninitialized, while the latter invariably zero-initializes its `i`. The second difference, seen in `I1b` versus `S1b`, is that the former is explicitly not initialized whereas it is always implicitly initialized in the latter.

Note that, although every literal type needs to have a way to construct it in a **constexpr** context, not *every* constructor of a literal type needs to be **constexpr**; see *Literal types (defined)* on page 25.

6. **Braced initialization**, while not always incurring constructor invocation, can still be done during compile-time evaluation. This initialization must only involve operations that can be done during constant evaluation, but, unlike a hand-written constructor, it will (a) first **zero-initialize** all members and (b) skip a (possibly deleted or non**constexpr**) **trivial** default constructor.[11].

   Braced initialization can produce surprising cases where a non**constexpr** constructor seems like it should be invoked but is instead skipped during braced or value initialization because it is trivial:

```
struct S1  // example with a nonconstexpr trivial default constructor
{
    int d_i;        // not initialized by S1()
    S1() = default; // trivial, nonconstexpr
};

static_assert( S1().d_i  == 0, "");  // OK, value initialization
static_assert( S1{}.d_i  == 0, "");  // OK, braced initialization
static_assert( S1{7}.d_i == 7, "");  // OK, braced (list) initialization
```

[11] A default constructor is trivial if (a) it is implicit, defaulted, or deleted; (b) all non**static** data members have trivial default constructors and no default member initializers; and (c) all base classes are non**virtual** and have trivial default constructors.

Braced initialization can even skip a deleted constructor (see Section 1.1."**??**" on page **??**) where regular initialization would not:

```
struct S2  // example with a deleted default constructor
{
    int d_i;       // not initialized
    S2() = delete; // trivial
};

static_assert( S2().d_i  == 0, "");  // Error, invokes deleted constructor
static_assert( S2{}.d_i  == 0, "");  // OK, braced initialization
static_assert( S2{7}.d_i == 7, "");  // OK, braced (list) initialization
```

Finally, it is important to note that this form of braced initialization is not restricted to just aggregates and requires only that the default constructor be trivial[12]:

```
struct S3 {  // example of a non*aggregate* type
    int d_i;              // not initialized
    S3() = default;       // trivial
    S3(int i) : d_i(i) {} // not an aggregate
};

static_assert( S3().d_i  == 0, "");  // OK, value initialization
static_assert( S3{}.d_i  == 0, "");  // OK, braced initialization
static_assert( S3{7}.d_i == 7, "");  // Error, nonconstexpr constructor
```

7. For a **union**, exactly one of its data members must be initialized with (1) a default initializer that is a *constant expression* (see Section 2.1."**??**" on page **??**), (2) a **constexpr** constructor, or (3) braced initialization that picks a member that can be initialized in a **constexpr** context:

```
// unions having no explicit constructors
union U0 { bool b;      char c;       };  // OK, neither member initialized
union U1 { bool b = 0;  char c;       };  // OK, first      "       "
union U2 { bool b;      char c = 'A'; };  // OK, second     "       "
union U3 { bool b = 0;  char c = 'A'; };  // Error, multiple members initialized

// unions having constexpr constructors
union U4 { bool b; char c;        constexpr U4() { } };  // Error, uninitialized
union U5 { bool b; char c = 'A';  constexpr U5() { } };              // OK
union U6 { bool b; char c;        constexpr U6() : c('A') { } };     // OK
union U7 { bool b; char c;        constexpr U7(bool v) : b(v) { } };  // OK
```

[12]The original intent was clearly to enable any initialization that involved only operations that could be done at compile time to be a valid initialization for a literal type. This was originally noted by Alisdair Meredith in CWG issue 644 (see **?**), which was inadvertently undone by mistakenly generalizing the solution to just aggregates with the resolution of CWG issue 981 (see **?**) and is pending a final resolution when wording is adopted for CWG issue 1452 (see **?**). Even with the actual wording not allowing this form of initialization to be considered enough to make a type a literal type, all current compiler implementations have consistently adopted support for this interpretation and we are simply waiting for the Standard to catch up to existing practice.

```
struct S                          // S is a *literal type*.
{
    U0 u0{};                      // *braced*  initialized
    U1 u1; U2 u2; U5 u5; U6 u6;   // *default* initialized
    U7 u7;                        // *value*   initialized
    constexpr S() : u7(0) { }     // OK, all members are initialized.
};

constexpr int test(S t) { return 0; }  // OK, confirms S is a *literal type*
```

The example code above illustrates various ways in which unions (e.g., `U0`–`U2` and `U5`–`U7`) can be used that allow them to be initialized by a **`constexpr`** constructor (e.g., `S()`). The existence of at least one (non*copy* non*move*) **`constexpr`** constructor implies that the class (e.g., `S`) comprising these unions is a literal type, which we have confirmed using the C++11 **interface test** idiom; see **??** on page **??**.[**AUs: There is no section called "Parameters and return types" Did you mean "constexpr-function parameter and return types"?**]

8. If the constructor *delegates* to another constructor in the same class (see Section 1.1."**??**" on page **??**), that target constructor must be **`constexpr`**:

```
struct C0  // BAD IDEA: Only the default constructor is constexpr.
{
              C0(int)      { }  // Ok, but not declared constexpr
    constexpr C0() : C0(0) { }  // Error, delegating to nonconstexpr ctor
};

struct C1  // GOOD IDEA: Both default and value constructor are constexpr.
{
    constexpr C1(int)      { }  // Ok, declared constexpr
    constexpr C1() : C1(0) { }  // Ok, delegating to constexpr constructor
};
```

9. When initializing data members of a class (e.g., `S` below), any nonconstructor functions needed for implicitly converting the type of the initializing expression (e.g., `V` in the code snippet below) to that of data member (e.g., **`int`** or **`double`**) must also be **`constexpr`**:

```
struct V
{
    int v;
              operator    int() const { return v; }  // implicit conversion
    constexpr operator double() const { return v; }  // implicit conversion
};

struct S
{
    int i; double d;  // A constexpr constructor must initialize both members.
```

```
    constexpr S(const V& x, double y) : i(x), d(y) { }  // Error, the needed
        // int implicit conversion is *not* declared constexpr.

    constexpr S(int x, const V& y) : i(x), d(y) { }    // OK, the needed
        // double implicit conversion *is* declared constexpr.
};
```

expr-function-templates

constexpr function templates

Function templates, member function templates, and constructor templates can all be declared **constexpr** too and more liberally than for nontemplated entities. That is, if a particular explicit specialization of such a template doesn't meet the requirements of a **constexpr** function, member function, or constructor, it will not be invocable at compile time.[13] For example, consider a function template, `sizeOf`, that is **constexpr** only if its argument type, `T`, is a literal type:

```
template <typename T> constexpr int sizeOf(T t) { return sizeof(t); }
    // This function is constexpr *only* if T is a *literal type*.

struct S0 { int i;           S0() : i(0) { } };  // S0 is *not* a *literal type*.
struct S1 { int i; constexpr S1() : i(0) { } };  // S1 *is* a *literal type*.

int a[sizeOf(int())];  // OK,    int *is*     a *literal type*.
int b[sizeOf( S0())];  // Error,  S0 is *not* a *literal type*.
int c[sizeOf( S1())];  // OK,     S1 *is*     a *literal type*.
```

If no specialization of such a function template would yield a **constexpr** function, then the program is IFNDR. For example, if this same function template were implemented with a function body consisting of more than just a single **return** statement, it would be ill formed:

```
template <typename T>
constexpr int badSizeOf(T t) { const int s = sizeof(t); return s; }
    // This constexpr function template is *IFNDR*.
```

Most compilers, when compiling such a specialization for runtime use, will not attempt to determine if the **constexpr** would ever be valid. When invoked with arguments that are themselves constant expressions (such as a temporary of literal type), they do, however, often detect this ill formed nature and report the error:

```
int d[badSizeOf(S1())];  // Error, badSizeOf<S1>(S1) body not return statement
int e[badSizeOf(S0())];  // Error, badSizeOf<S0>(S0) body not return statement
int f = badSizeOf(S1()); // Oops, same issue but *might* work on some compilers
int g = badSizeOf(S0()); // Oops, same issue but *often* works without warnings
```

It is important to understand that each of the four statements in the code snippet above are ill formed because the `badSizeOf` function template is itself ill formed. Although the

[13] A specialization that cannot be evaluated at compile time is, however, still considered **constexpr**. This is not readily observable but does enable some generic code to remain well formed as long as the particular specializations are not actually required to be evaluated at compile time. This rule was adopted with the resolution of CWG issue 1358 (see **?**).

compiler is not required to diagnose the general case, once we try to use an explicit instantiation of it in a **constexpr** context (e.g., `d` or `e`), it is mandatory that the supplied argument be used to determine the value of the constant expression or fail trying. When used in a non**constexpr** context (e.g., `f` or `g`), whether the compiler fails, warns, or proceeds is a matter of **quality of implementation (QoI)**.

eter-and-return-types

constexpr-function parameter and return types

At this point, we arrive at what is perhaps the most confounding part of the seemingly *cyclical* definition of **constexpr** functions: A function cannot be declared **constexpr** unless the return type and every parameter of that function satisfies the criteria for being a literal type; a literal type is the category of types whose objects are permitted to be created and destroyed when evaluating a constant expression:

```
struct  Lt { int v; constexpr  Lt() : v(0) { } };  // literal type
struct Nlt { int v;           Nlt() : v(0) { } };  // nonliteral type

          Lt  f1() { return Lt();  }  // OK, no issues
constexpr Lt  f2() { return Lt();  }  // OK, returning literal type
          Nlt f3() { return Nlt(); }  // Ok, function is nonconstexpr
constexpr Nlt f4() { return Nlt(); }  // Error, constexpr returning nonliteral

          int g1(Lt  x) { return x.v; }  // OK, no issues
constexpr int g2(Lt  x) { return x.v; }  // OK, parameter is a literal type
          int g3(Nlt x) { return x.v; }  // OK, function is nonconstexpr
constexpr int g4(Nlt x) { return x.v; }  // Error, constexpr taking nonliteral
```

Consider that all *pointer* and *reference* types — being *built-in types* — are literal types and therefore can appear in the interface of a **constexpr** function irrespective of whether they point to a literal type:

```
constexpr int h1(Lt*  p) { return p->v; }  // OK, parameter is a literal type
constexpr int h2(Nlt* p) { return p->v; }  // OK,     "      "  "    "      "
constexpr int h3(Lt&  r) { return r.v; }   // OK,     "      "  "    "      "
constexpr int h4(Nlt& r) { return r.v; }   // OK,     "      "  "    "      "
```

However, note that, because it is not possible to construct an object of nonliteral type at compile time, there is no way to invoke `h2` or `h4` as part of a constant expression since the access of the member `v` in all of the above functions requires an already created object to exist. Pointers and references to nonliteral types that do not access those types can still be used:

```
Nlt arr[17];
constexpr Nlt& arr_0 = arr[0];                // OK, initializing a reference
constexpr Nlt *arr_0_ptr = &arr[0];           // OK, taking an address
constexpr Nlt& arr_0_ptr_deref = *arr_0_ptr;  // OK, dereferencing but not using
static_assert(&arr[17] - &arr[4] == 13,"");   // OK, pointer arithmetic

constexpr int arr_0_v = arr_0.v;              // Error, arr[0] is not usable.
constexpr int arr_0_ptr_v = arr_0_ptr->v;     // Error,    "     "   "    "
```

literal-types-(defined)

Literal types (defined)

Until now, we've discussed many ways in which understanding which types are literal types is important to understand what can and cannot be done during compile-time evaluation. We now elucidate how the language defines a literal type and, as such, how they are usable in two primary use cases:

- Literal types are eligible to be created and destroyed during the evaluation of a *constant expression*.
- Literal types are suitable to be used in the *interface* of a **constexpr** function, either as the return type or as a parameter type.

The criteria for determining whether a given type is a literal type can be divided into six parts:

1. Every scalar type is a literal type. Scalar types include all fundamental arithmetic (integral and floating point) types, all enumerated types, and all pointer types.

int	**int** is a *literal type*.
double	**double** is a *literal type*.
char*	**char*** is a *literal type*.
enum E { e_A }	E is a *literal type*.
struct S { S(); };	S is *not* a *literal type*.
S*	S* *is* a *literal type* (even though S is *not*).

   Note that a pointer is *always* a literal type — even when it points to a type that itself is *not* a literal type.

2. Just as with pointers, every reference type is a literal type irrespective of whether the type to which it refers is itself a literal type.

int&	**int&** is a *literal type*
S&	S& is a *literal type* (even if S is not).
S&&	S&& is a *literal type*.

3. A **class**, **struct**, or **union** is a literal type if it meets each of these four requirements:
   (a) It has a trivial destructor.[14]
   (b) Each non**static** data member is a non**volatile** literal type.[15]
   (c) Each base class is a literal type.
   (d) There is some way to initialize an object of the type during constant evaluation; either (a) it is an **aggregate type**, (b) it can be braced initialized in a **constexpr** context, or (c) it has at least one **constexpr** constructor (possibly a template) that is not a *copy* or *move* constructor:

[14] As of C++20, a destructor can be declared **constexpr** and even both **virtual** and **constexpr**.
[15] In C++17, this restriction is relaxed: For a **union** to be a literal type, only one, rather than all, of its non**static** data members needs to be of a non**volatile** literal type.

```
#include <string>  // std::string
struct LiteralUDT
{
    static std::string s_cache;
        // OK, static data member can have a nonliteral type.

    int d_datum;
        // OK, nonstatic data member of nonvolatile *literal type*

    constexpr LiteralUDT(int datum) : d_datum(datum) { }
        // OK, constructor is constexpr.

    // constexpr ~LiteralUDT() { }  // not permitted until C++20
        // no need to define; implicitly-generated destructor is trivial
};

struct LiteralAggregate
{
    int d_value1;
    int d_value2;
};

struct LiteralBraceInitializable
{
    int d_value1;
    int d_value2;
    LiteralBraceInitializable() = default; // trivial default constructor
    LiteralBraceInitializable(int v1, int v2)
        : d_value1(v1), d_value2(v2) { }   // not an aggregate
};

union LiteralUnion
{
    int   d_x;  // OK, int is a *literal type*.
    float d_y;  // OK, float is a *literal type*.
};
```

4. A cv-qualified literal type is also a literal type.[16]

const int	is a *literal type*
volatile int	is a *literal type*
const volatile int	is a *literal type*
const `LiteralUDT`	is a *literal type* (since `LiteralUDT` is)

5. Arrays of objects of literal type are also literal types:

[16] Note that cv-qualified scalar types are still scalar types, and cv-qualified **class** types were noted as being literal types in a defect report that resolved CWG issue 1951 (see **?**).

```
char a[5];  // An array of **scalar type** (e.g., char) is a **literal type**.

struct { int i; bool b; } b[7];
    // An array of **aggregate type** is a **literal type**.
```

6. In C++14 and thereafter, **void** (and thus cv-qualified **void**) is also a literal type, thereby enabling functions that return **void**:

```
constexpr const volatile void f() { }  // OK, in C++14
```

The overarching goal of this six-part definition of what constitutes a literal type is to capture those types that might be *eligible* to be created and destroyed during evaluation of a *constant expression*. This definition does not, however, guarantee that every literal type satisfying the above criteria will necessarily be constructible in a constant expression, let alone in a meaningful way.

- A user-defined literal type is not required to have any **constexpr** member functions or publicly accessible members. It is quite possible that the only thing one might be able to do with a user-defined literal type as part of a constant expression is to create it:

```
class C { };  // C is a *valueless literal type*.

int a[(C(), 5)];  // OK, create an array of five int objects.
```

  Such "barely literal" types — though severely limited in their usefulness in constant expressions — do allow for useful compile-time initialization of **constexpr** variables in C++14 (see Section 2.1."**??**" on page **??**).

- The requirement to have at least one **constexpr** constructor that is not a *copy* or *move* constructor is just that — to have one. There is no requirement that such a constructor be invocable at compile time (e.g., it could be declared **private**) or even that it be defined; in fact, a deleted constructor (see Section 1.1."**??**" on page **??**) satisfies the requirement:

```
struct UselessLiteralType
{
    constexpr UselessLiteralType() = delete;
};
```

- Many uses of literal types in **constexpr** functions will require additional **constexpr** functions to be defined (i.e., not *deleted*), such as a *move* or *copy* constructor:

```
struct Lt  // literal type having nonconstexpr copy constructor
{
    constexpr Lt(int c) { }  // valid constexpr value constructor
    Lt(const Lt& ) { }       // nonconstexpr copy constructor
};
```

```
constexpr int processByValue(Lt t) { return 0; }  // valid constexpr function

static_assert(processByValue(Lt(7)) == 0, "");
    // Error, but OK (due to elided copy) on some platforms

constexpr Lt s{7};  // braced-initialized aggregate

static_assert(processByValue(s) == 0, "");  // Error, nonconstexpr copy ctor
```

In the code example above, we have an *aggregate* literal type**, `Lt`**, for which we have explicitly declared a non**constexpr** copy constructor. We then defined a valid **constexpr** function, `processByValue`, taking an `Lt` (by value) as its only argument. Invoking the function by constructing a object of `Lt` from a literal **int** value enables the compiler to elide the copy. Platforms where the copy is elided might allow this evaluation at compile time, while on others there will be an error. When we consider using an independently constructed **constexpr** variable, `s`, the `copy` can no longer be elided, and, since the copy constructor is declared explicitly to be non**constexpr**, the compile-time assertion fails to compile on all platforms; see Section 2.1."**??**" on page **??**.

- Although a pointer or reference is always (by definition) a *literal type*, if the type being pointed to is not itself a literal type, then the referenced object cannot be used.

tifying-literal-types

Identifying literal types

Knowing what is and what is not a literal type is not always obvious (to say the least) given all the various rules we have covered. Having a concrete way *other* than becoming a language lawyer and interpreting the full standard definition can be immensely valuable during development, especially when trying to prototype a facility that you intend to be usable at compile time. In this subsection, we identify means for ensuring that a type is a literal type and, often more importantly for a user, identifying if a type is a ***usable* literal type**.

1. Only literal types can be used in the interface of a **constexpr** function (i.e., either as the return type or as a parameter type), and any literal type can be used in the interface of such a function. The first approach one might take to determine if a given type is a literal type would be to define a function that returns the given type *by value*. This has the downside of requiring that the type in question also be *copyable* (or at least *movable*, see Section 2.1."**??**" on page **??**)[17]:

```
struct LiteralType    { constexpr LiteralType(int i)    {}         };
struct NonLiteralType {           NonLiteralType(int i) {}         };
struct NonMovableType { NonMovableType(NonMovableType&&) = delete; };

constexpr LiteralType    f(int i) { return LiteralType(i);    }  // OK
constexpr NonLiteralType g(int i) { return NonLiteralType(i); }  // Error
constexpr NonMovableType h()      { return NonMovableType{};  }  // Error
```

[17] As of C++17, the requirement that the type in question be *copyable* or *movable* to return it as a *prvalue* is removed; see Section 2.1."**??**" on page **??**.

In the above example, `NonMovableType` is a literal type but is not movable (or copyable), so it cannot be the return type of a function. Passing the type as a *by-value* parameter works more reliably — and even reliably identifies non*copyable*, non*movable* literal types**:

```
constexpr int test(LiteralType t)    { return 0; } // OK
constexpr int test(NonLiteralType t) { return 0; } // Error
constexpr int test(NonMovableType t) { return 0; } // OK
```

This approach is appealing in that it provides a general way for a programmer to query the compiler whether it considers a given type, `S`, *as a whole* to be a literal type and can be succinctly written[18]:

```
constexpr int test(S) { return 0; }  // compiles only if S is a *literal type*
```

Note that all of these tests require providing a function body, since compilers will validate that the declaration of the function is valid for a **`constexpr`** function only when they are processing the *definition* of the function. A declaration without a body will not produce the expected error for *non-literal-type* parameters and return types:

```
constexpr NonLiteralType quietly(NonLiteralType t);  // OK, declaration only
constexpr NonLiteralType quietly(NonLiteralType t) { return t; }  // Error
```

Finally, the C++11 Standard Library also provides a type trait — `std::is_literal_type` — that attempts to serve a similar purpose[19]:

```
#include <type_traits>  // std::is_literal_type
static_assert( std::is_literal_type<LiteralType>::value, "");     // OK
static_assert(!std::is_literal_type<NonLiteralType>::value, "");  // OK
```

Given the effectiveness of the simple **`constexpr`** interface test idiom illustrated in the code snippet above, any use of `std::is_literal_type` seems dubious.

The important take-away from this section is that we can use a trivial test in C++11 (made even more trivial in C++14) to find out if the compiler deems that a given type is a literal type. Much of the research done to understand and delineate this feature was made possible only through extensive use of this interface test idiom.

2. Often, simply *being* a literal type is not the only criterion that a developer is interested in. In order to ensure that a type under development is meaningful in a compile-time facility, it becomes imperative that we can confirm that objects of a given type can actually be constructed at compile-time. This confirmation requires identifying a

[18] As of C++14, this utility could be written as `template<typename S> constexpr void test(S) { }`, returning **`void`** and omitting the **`return`** statement entirely.

[19] Note that the `std::is_literal_type` trait is deprecated in C++17 and removed in C++20. The rationale is stated in **?**:

> The `is_literal_type` trait offers negligible value to generic code, as what is really needed is the ability to know that a specific construction would produce constant initialization. The core term of a literal type having at least one **`constexpr`** constructor is too weak to be used meaningfully.

particular form of initialization and corresponding **witness arguments** that should allow a user-defined type to assume a valid compile-time value. For this example, we can use the interface test to help prove that our class (e.g., `Lt`) is a literal type:

```
class Lt  // An object of this type can appear as part of a **constant expression**.
{
    int d_value;

public:
    constexpr Lt(int i) : d_value(i != 75033 ? throw 0 : i) { }  // OK
};

constexpr int checkLiteral(Lt) { return 0; }  // OK, *literal type*
```

Proving that `Lt` (in the code example above) is a *usable* literal type next involves choosing a **`constexpr`** constructor (e.g., **`Lt(int)`**), selecting appropriate witness arguments (e.g., `75033`), and then using the result in a constant expression. The compiler will indicate (by producing an error) if our type cannot be constructed at compile time:

```
char a[(Lt(75033), 1)];               // OK, usable in constant expr
static_assert((Lt(75033), true), ""); // OK,    "    "     "       "
```

For types that are not *usable* literal types, there will be no such proof. When a particular constructor that is *explicitly* declared to be **`constexpr`** has no sequence of witness arguments that can be used to prove that the type is usable, the constructor (and any program in which it resides) is IFNDR. Forcing the compiler to perform such a proof in general — even if such were possible — would not be a wise use of compile-time compute resources. Hence, compilers will generally not diagnose the ill formed constructor and instead simply produce an error on each attempt to provide a set of **witnesses** for a literal type that fails to be usable at compile time:

```
int a = 1, b = 2;  // a and b are not constexpr.

class PathologicalType  // ill formed, no diagnostic required
{
    int d_value;

public:
    constexpr PathologicalType(int i)
        : d_value( (i <  2) ? a
                 : (i >= 2) ? b
                 : (i * 2) ) { }
};
```

The compiler is unlikely to have logic to discover that there is no way to invoke the above **`constexpr`** constructor as part of the evaluation of a constant expression; the constructor is considered ill formed, but no diagnostic is likely to be produced. Supplying any witness arguments, however, will force the compiler to evaluate the constructor and discover that no *particular* invocation is valid:

```
static_assert((PathologicalType(1),true), "");  // Error, a is not constexpr.
static_assert((PathologicalType(2),true), "");  // Error, b is not constexpr.
static_assert((PathologicalType(3),true), "");  // Error, b is not constexpr.
```

compile-time-evaluation

Compile-time evaluation

All of the restrictions on the constructs that are valid in a **constexpr** function exist to enable the portable evaluation of such functions at compile time. Appreciating this motivation requires an understanding of compile-time calculations in general and constant expressions in particular.

First, a constant expression is *required* in very specific contexts:

- Any arguments to **static_assert**, **noexcept**-exception specifications, and the **alignas** specifier
- The size of a built-in array
- The expression for a **case** label in a **switch** statement
- The initializer for an enumerator
- The length of a bit field
- Nontype template parameters
- The initializer of a **constexpr** variable (see Section 2.1."**??**" on page **??**)

Computing the value of expressions in these contexts requires that all of their subexpressions be known and evaluable at compile time, except those that are short-circuited by the logical *or* operator (`||`), the logical *and* operator (`&&`), and the *ternary* operator (`?:`):

```
constexpr int f(int x) { return x || (throw x, 1); }
constexpr int g(int x) { return x && (throw x, 1); }
constexpr int h(int x) { return x ? 1 : throw x; }

static_assert(f(true), "");    // OK, throw x is never evaluated.
static_assert(!g(false), "");  // OK,     "     "    "      "
static_assert(h(true), "");    // OK,     "     "    "      "
```

Note that the **controlling constant expression** for the preprocessor directives **#if** and **#elif**, while similar to general constant expressions, are computed at a time before any functions (**constexpr** or not) are even parsed, and all but a fixed set of predefined identifiers (e.g., `defined` and **true**) are replaced with macro-expansions or `0` before the resulting arithmetic expression is evaluated. This small subset of other compile-time evaluation facilities in the language cannot invoke any user-defined functions. Consequently, **constexpr** functions cannot be invoked as part of the controlling constant expression for preprocessor directives.

Second, the Standard identifies a clear set of operations that are not available for use in constant expressions and, therefore, cannot be relied upon for compile-time evaluation:

- Throwing an exception.
- Invoking the **new** and **delete** operators.
- Invoking a lambda function.
- Any operation that depends on runtime polymorphism, such as **dynamic_cast**, **typeid** on a polymorphic type, or invoking a virtual function, which cannot be **constexpr**.
- Using **reinterpret_cast**.
- Any operation that modifies an object (increment, decrement, and assignment), including function parameters, member variables, and global variables.
- Any operation having *undefined behavior* such as integer overflow, dereferencing **nullptr**, or indexing outside the bounds of an array.
- Invoking a non-**constexpr** function or constructor, or a **constexpr** function whose definition has not yet been seen.

Note that being marked **constexpr** enables a function to be evaluated *at compile time* only if (1) the argument values are constant expressions known before the function is evaluated and (2) no operations performed when invoking the function with those arguments involve any of the excluded ones listed above.

Global variables can be used in a **constexpr** function only if they are (1) *non***volatile const** objects of *integral* or *enumerated* type that are initialized by a constant expression (generally treated as **constexpr** even if only marked as **const**), or (2) **constexpr** objects of literal type; see *Literal types (defined)* on page 25 and Section 2.1."**??**" on page **??**. In either case, any **constexpr** global object used within a **constexpr** function must be initialized with a constant expression prior to the definition of the function. C++14[20] relaxes some of these restrictions (see Section 2.2."**??**" on page **??**).

Use Cases

use-cases

A better alternative to function-like macros

-function-like-macros

Computations that are useful both at run time and compile time and/or that must be inlined for performance reasons were typically implemented using preprocessor macros. For instance, consider the task of converting mebibytes to bytes:

```
#define MEBIBYTES_TO_BYTES(mebibytes) (mebibytes) * 1024 * 1024
```

The macro above can be used in contexts where both a constant expression is required and the input is known only during program execution:

```
#include <vector>  // std::vector
void example0(std::size_t input)
{
    char fixedBuffer[MEBIBYTES_TO_BYTES(2)];  // compile-time constant
```

[20] C++17 and C++20 each further relax these restrictions.

```
    std::vector<char> dynamicBuffer;
    dynamicBuffer.resize(MEBIBYTES_TO_BYTES(input));  // usable at run time
}
```

While a single-line macro with a reasonably unique (and long) name like `MEBIBYTES_TO_BYTES` is unlikely to cause any problems in practice, it harbors all the disadvantages macros have compared to regular functions. Macro names are not scoped; hence, they are subject to global name collisions. There is no well-defined input and output type and thus no type safety. Perhaps most tellingly, the lack of expression safety makes writing even simple macros tricky; a common error is to forget, e.g., the `()` around `mebibytes` in the implementation of `MEBIBYTES_TO_BYTES`, resulting in an unintended result if applied to a non-trivial expression such as `MEBIBYTES_TO_BYTES(2+2)` — yielding a value of `2 + 2 * 1024 * 1024 = 2097154` without the `()` and the intended value of `(2 + 2) * 1024 * 1024 = 4194304` with them. The generally unstructured and unhygienic nature of macros has led to significant language evolution aimed at supplanting their use with proper language features where practicable.[21]

A single **constexpr** function is sufficient to replace the `MEBIBYTES_TO_BYTES` macro, avoiding the aforementioned disadvantages without any additional runtime overhead:

```
constexpr std::size_t mebibytesToBytes(std::size_t mebibytes)
{
    return mebibytes * 1024 * 1024;
}

void example1(std::size_t input)
{
    char fixedBuffer[mebibytesToBytes(2)];
        // OK, guaranteed to be invocable at compile time

    std::vector<char> dynamicBuffer;
    dynamicBuffer.resize(mebibytesToBytes(input));
        // OK, can also be invoked at run time
}
```

Compile-time string traversal

e-time-string-traversal

Beyond simple numeric calculations, many compile-time libraries may need to accept strings as input and manipulate them in various ways. Applications can range from simply pre-calculating string-related values to powerful compile-time regular-expression libraries.[22] To begin, we will consider the simplest of string operations: calculating the length of a string. An initial implementation might attempt to leverage the type of a string constant (array of **char**) with a template:

```
#include <cstddef>  // std::size_t
```

[21] We are not suggesting that macros have no place in the ecosystem; in fact, many of the language features developed for C+++ — not the least of which were templates and, more recently, contract checks — were initially prototyped using preprocessor macros.

[22] See **?** for one example of how far such techniques can evolve and might potentially be incorporated into a future Standard Library release.

```
template <std::size_t N>
constexpr std::size_t constStrlenLit(const char (&lit)[N])
{
    return N - 1;
}
static_assert(constStrlenLit("hello") == 5, "");  // OK
```

This approach, however, fails when attempting to apply it to any number of other ways in which a variable might contain a compile-time or runtime string constant:

```
constexpr const char* hw1     = "hello";
char                  hw2[20] = "hello";
const char*           hw3     = hw2;

static_assert(constStrlenLit(hw1) == 5, "");  // Error, hw1 not a char[N]

std::size_t len2 = constStrlenLit(hw2);  // Bug, returns 19
std::size_t len3 = constStrlenLit(hw3);  // Error, hw3 not a char[N]
```

The type-based approach is clearly deficient. A better approach is simply to loop over the characters in the string, counting them until we find the terminating `\0` character. Here, we'll take the liberty of illustrating the simpler solution (using local variables and loops) that is available with the relaxed rules for **constexpr** functions in C++14 alongside the recursive solution that works in C++11; see Section 2.2."**??**" on page **??**:

```
constexpr std::size_t constStrlen(const char* str)
{
#if __cplusplus > 201103L
    const char *strEnd = str;
    while (*strEnd) ++strEnd;
    return strEnd - str;
#else
    return (str[0] == '\0') ? 0 : 1 + constStrlen(str + 1);
#endif
}

static_assert(constStrlen("hello") == 5, "");  // OK
static_assert(constStrlen(hw1)     == 5, "");  // OK

std::size_t len2b = constStrlen(hw2);  // OK, returns 5
std::size_t len3b = constStrlen(hw3);  // OK, returns 5
```

With this most basic function implemented, let's move on to the more interesting problem of counting the number of lowercase letters in a string in such a way that it can be evaluated at compile time if the string is a constant expression. We'll need a simple helper function that determines if a given **char** is a lowercase letter:

```
constexpr bool isLowercase(char c)
    // Return true if c is a lowercase *ASCII* letter, and false otherwise.
{
```

```
    return 'a' <= c && c <= 'z';  // true if c is in *ASCII* range a to 'z'
}
```

Note that we are using a simplistic definition here that is designed to handle only the *ASCII* letters `a` through `z`; significantly more work would be required to handle other character sets or locales at compile time. Unfortunately, the `std::islower` function inherited from C is not **`constexpr`**.

Now we can apply a very similar construct to what we used for `constStrlen` to count the number of lowercase letters in a string:

```
constexpr std::size_t countLowercase(const char* str)
{
    return (str[0] == '\0') ? 0 : isLowercase(str[0]) + countLowercase(str + 1);
}
```

Now we have a function that will count the lowercase letters in a string at either compile time or run time for *any* null-terminated string:

```
static_assert(countLowercase("") == 0, "");
static_assert(countLowercase("HELLO, WORLD") == 0, "");
static_assert(countLowercase("Hello, World") == 8, "");

#include <cassert>  // standard C assert macro
void test1()
{
     const char *p1 = "";              assert(countLowercase(p1) == 0);
     const char *p2 = "HELLO, WORLD";  assert(countLowercase(p2) == 0);
     const char *p3 = "Hello, World";  assert(countLowercase(p3) == 8);
}
```

The first three invocations of `countLowercase`, in the code snippet above, illustrate that, when given a **`constexpr`** argument, it can compute the correct result at compile time. The other three invocations show that `countLowercase` can be invoked on non**`constexpr`** strings and compute the correct results at run time.

Even with the seemingly austere restrictions on C++11 **`constexpr`** function bodies, much of the power of the language is still available at compile time. For example, counting values in an array that match a function predicate can be converted to a **`constexpr`** function template in the same way we might do so with a runtime-only template:

```
template <typename T, typename F>
constexpr std::size_t countIf(T* arr, std::size_t len, const F& func)
{
    return (len==0) ? 0 : (func(arr[0]) ? 1 : 0) + countIf(arr+1,len-1,func);
}
```

In just one dense line, `countIf` recursively determines if the current length, `len`, is 0 and, if not, whether the first element satisfies the predicate, `func`. If so, `1` is added to the result of recursively invoking `countIf` for the rest of the elements in `arr`.

This `countIf` function template can now be used at compile time with a **`constexpr`** function pointer to produce a more modern-looking version of our `countLowercase` function:

```
constexpr std::size_t countLowercase(const char* str)
{
    return countIf(str, constStrlen(str), isLowercase);
}
```

Rather than a null-terminated array of **char**, we might want a more flexible string representation consisting of a **class** containing a **const char*** pointing to the start of a sequence of **char**s and a `std::size_t` holding the length of the sequence.[23] We can define such a class as a literal type even in C++11:

```
#include <stdexcept>  // std::out_of_range

class ConstStringView
{
    const char* d_string_p;  // address of the string supplied at construction
    std::size_t d_length;    // length  "  "     "       "     "       "
public:
    constexpr ConstStringView(const char* str)
    : d_string_p(str)
    , d_length(constStrlen(str)) {}

    constexpr ConstStringView(const char* str, std::size_t length)
    : d_string_p(str)
    , d_length(length) {}

    constexpr char operator[](std::size_t n) const
    {
        return n < d_length ? d_string_p[n] : throw std::out_of_range("");
    }

    constexpr const char *data() const { return d_string_p; }
    constexpr std::size_t length() const { return d_length; }

    constexpr const char* begin() const { return d_string_p; }
    constexpr const char* end()  const { return d_string_p + d_length; }
};
```

The `ConstStringView` class shown above provides some basic functionality to inspect and pass around the contents of a string constant at compile time. The implicitly declared **constexpr** copy constructor of this literal type allows us to overload our `countIf` function template and `countLowercase` function to take a `ConstStringView` by value:

```
template <typename F>
constexpr std::size_t countIf(ConstStringView sv, const F& func)
{
    return countIf(sv.data(), sv.length(), func);
}

constexpr std::size_t countLowercase(ConstStringView sv)
```

[23] C++17's `std::string_view` is an example of such string-related utility functionality.

```
{
    return countIf(sv, isLowercase);
}
```

Thanks to the implicit-conversion constructors, all of the earlier **static_assert** statements that were used with previous `countLowercase` implementations work with this one as well, and we gain the ability to further use `ConstStringView` as a **vocabulary type** for our **constexpr** functions.

f-data-at-compile-time

Precomputing tables of data at compile time

Often, compile-time evaluation through the use of **constexpr** functions can be used to replace otherwise complex template metaprogramming or preprocessor tricks. While yielding more readable and more maintainable source code, **constexpr** functions also enable useful computations that previously were simply not practicable at compile time.

Calculating single values and using them at compile time is straightforward. Storing such values to use at run time can be done with a **constexpr** variable; see Section 2.1."**??**" on page **??**. Calculating many values and then using them at run time benefits from other modern language features, in particular variadic templates (see Section 2.1."**??**" on page **??**).

Consider a part of a date and time library that provides utilities to deal with modern timestamps of type `std::time_t` — an integer type expressing a number of seconds since some point in time, e.g., the **POSIX epoch**, midnight on January 1, 1970. An important tool in this library would be a function to determine the year of a given timestamp:

```
#include <ctime>  // std::time_t

int yearOfTimestamp(std::time_t timestamp);
    // Return the year of the specified timestamp. The behavior is undefined
    // if timestamp < 0.
```

Among other features, this library would provide a number of constants, both for its internal use as well as for direct client use. These could be implemented as enumerations, as integral constants at namespace scope, or as static members of a **struct** or **class**. Since we will be leveraging them within **constexpr** functions, we will also illustrate making use of constexpr variables here:

```
// constants defining the date and time of the epoch
constexpr int k_EPOCH_YEAR  = 1970;
constexpr int k_EPOCH_MONTH = 1;
constexpr int k_EPOCH_DAY   = 1;

// constants defining conversion ratios between various time units
constexpr std::time_t k_SECONDS_PER_MINUTE = 60;
constexpr std::time_t k_SECONDS_PER_HOUR   = 60  * k_SECONDS_PER_MINUTE;
constexpr std::time_t k_SECONDS_PER_DAY    = 24  * k_SECONDS_PER_HOUR;
constexpr std::time_t k_SECONDS_PER_YEAR   = 365 * k_SECONDS_PER_DAY;
static_assert( 31536000L == k_SECONDS_PER_YEAR, "");
```

For practical reasons related to the limits that compilers put on template expansion and **constexpr** expression evaluation, this library will support only a moderate number of future

years:

```
// constant defining the largest year supported by our library
constexpr int k_MAX_YEAR = 2200;
```

To begin implementing `yearOfTimestamp`, it helps to start with an implementation of a solution to the reverse problem — calculating the timestamp of the start of each year, which requires an adjustment to account for leap days:

```
constexpr std::time_t numLeapYearsSinceEpoch(int year)
{
    return (year         / 4) - (year         / 100) + (year         / 400)
        - ((k_EPOCH_YEAR / 4) - (k_EPOCH_YEAR / 100) + (k_EPOCH_YEAR / 400));
}

constexpr std::time_t startOfYear(int year)
    // Return the number of seconds between the epoch and the start of the
    // specified year.  The behavior is undefined if year < 1970 or
    // year > k_MAX_YEAR.
{
    return (year - k_EPOCH_YEAR) * k_SECONDS_PER_YEAR
         + numLeapYearsSinceEpoch(year - 1) * k_SECONDS_PER_DAY;
}
```

Given these tools, we could implement `yearOfTimestamp` naively with a simple loop:

```
int yearOfTimestamp(std::time_t timestamp)
{
    int year = k_EPOCH_YEAR;
    for (; timestamp > startOfYear(year + 1); ++year) {}
    return year;
}
```

This implementation, however, has algorithmically poor performance. While a closed-form solution to this problem is certainly possible, for expository purposes we will consider how we might, at compile time, build a lookup table of the results of `startOfYear` so that `yearOfTimestamp` can be implemented as a **binary search** on that table.

Populating a built-in array at compile time is feasible by manually writing each initializer, but a decidedly better option is to generate the sequence of numbers we want as a `std:array` where all we need is to provide the **`constexpr`** function that will take an index and produce the value we want stored at that location within the array. We will start by implementing the pieces needed to make a generic **`constexpr`** function for initializing `std::array` instances with the results of a **function object** applied to each index:

```
#include <array>  // std::array

template <typename T, std::size_t N, typename F>
constexpr std::array<T, N> generateArray(const F& func);
    // Return an array arr of size N such that arr[i] == func(i) for
    // each i in the half-open range [0, N).
```

The common idiom to do this initialization is to exploit a type that encodes indices as a variadic parameter pack (see Section 2.1."**??**" on page **??**), along with the help of some using aliases (see Section 1.1."**??**" on page **??**)[24]:

```
template <std::size_t...>
struct IndexSequence
{
    // This type serves as a compile-time container for the sequence of size_t
    // values that form its template parameter pack.
};

template <std::size_t N, std::size_t... Seq>
struct MakeSequenceHelper : public MakeSequenceHelper<N-1u, N-1u, Seq...>
{
    // This type is a metafunction to prepend a sequence of integers 0 to N-1
    // to the Seq... parameter pack by prepending N-1 to Seq... and
    // recursively instantiating itself.  The resulting integer sequence is
    // available in the type member inherited from the recursive instantiation.
    // The type member has type IndexSequence<FullSequence...>, where
    // FullSequence is the sequence of integers 0 .. N-1, Seq....
};

template <std::size_t ... Seq>
struct MakeSequenceHelper<0U, Seq...>
{
    // This partial specialization is the base case for the recursive
    // inheritance of MakeSequenceHelper.  The type member is an alias for
    // IndexSequence<Seq...>, where the Seq... parameter pack is typically
    // built up through recursive invocations of the MakeSequenceHelper
    // primary template.

    using type = IndexSequence<Seq...>;
};

template <std::size_t N>
using MakeIndexSequence = typename MakeSequenceHelper<N>::type;
    // alias for an IndexSequence<0 .. N-1> (or IndexSequence<> if N is 0)
```

To implement our array initializer, we will need another helper function that has, as a template argument, a variadic parameter pack of indices. To get this template parameter pack `std::size_t... I` deduced, our function has an unnamed argument of type `IndexSequence<I...>`. With this parameter pack in hand, we can then use a simple pack expansion expression and braced initialization to populate our `std::array` return value:

```
template <typename T, std::size_t... I, typename F>
constexpr std::array<T, sizeof...(I)> generateArrayImpl(const F& func,
                                                        IndexSequence<I...>)
```

[24]This idiom was, in fact, so common that it is available in the Standard Library as `std::index_sequence` in C++14 (see **?**). Note that this solution is not lightweight, so the Standard Library types are generally implemented using compiler intrinsics that make them usable for significantly larger values.

```
    // Return the results of calling F(i) for each i in the pack deduced as
    // the template parameter pack I.
{
    return { func(I)... };
}
```

The return statement in `generateArrayImpl` calls `func(I)` for each `I` in the range from 0 to the length of the returned `std::array`. The resulting pack of values is used to list initialize the return value of the function; see Section 2.1."??" on page ??.

Finally, our implementation of `generateArray` forwards `func` to `generateArrayImpl`, using `MakeIndexSequence` to generate an object of type `IndexSequence<0,...,N-1>`:

```
template <typename T, std::size_t N, typename F>
constexpr std::array<T, N> generateArray(const F& func)
{
    return generateArrayImpl<T>(func, MakeIndexSequence<N>());
}
```

With these tools in hand and a support function to offset the array index with the year, it is now simple to define an array that is initialized at compile time with an appropriate range of results from calls to `startOfYear`:

```
constexpr std::time_t startOfEpochYear(int epochYear)
{
    return startOfYear(k_EPOCH_YEAR + epochYear);
}

constexpr std::array<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR> k_YEAR_STARTS =
    generateArray<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR>(startOfEpochYear);

static_assert(k_YEAR_STARTS[0]  == startOfYear(1970),"");
static_assert(k_YEAR_STARTS[50] == startOfYear(2020),"");
```

With this table available for our use, the implementation of `yearOfTimestamp` becomes a simple application of `std::upper_bound` to perform a binary search on the sorted array of start-of-year timestamps[25]:

```
#include <algorithm>  // std::upper_bound
int yearOfTimestamp(std::time_t timestamp)
{
    std::size_t ndx = std::upper_bound(k_YEAR_STARTS.begin(),
                                       k_YEAR_STARTS.end(),
                                       timestamp)
                    - k_YEAR_STARTS.begin();
    return k_EPOCH_YEAR + ndx - 1;
}
```

[25]Among other improvements to language and library support for **constexpr** programming, C++20 added **constexpr** to many of the standard algorithms in `<algorithm>`, including `std::upper_bound`, which would make switching this implementation to be **constexpr** also trivial. Implementing a **constexpr** version of most algorithms in C++14 is, however, relatively simple (and in C++11 is still possible), so, given a need, providing **constexpr** versions of functions like this with less support from the Standard Library is straightforward.

When implementing a library of this sort, carefully making key decisions, such as whether to place the **constexpr** calculations in a header or to insulate them in an implementation file, is important; see *Potential Pitfalls — Overzealous use hurts* on page 43. When building tables such as this, it's also worth considering more classical alternatives, such as simply generating code using an external script. Such external approaches can yield significant reductions in compile time and improved insulation; see *Potential Pitfalls — One time is cheaper than compile time or run time* on page 44.

-pitfalls-constexprfunc

Potential Pitfalls

compile-time-evaluation

Low compiler limits on compile-time evaluation

A major restriction on compile-time evaluation, beyond the linguistic restrictions already discussed, is the set of **implementation-defined** limitations specific to the compiler. In particular, the Standard allows implementations to limit the following:

- *Maximum number of recursively nested* **constexpr** *function invocations*: The expected value for this limit is 512, and in practice that is the default for most implementations. While 512 may seem like a large call depth, C++11 **constexpr** functions must use recursion instead of iteration, making it easy to exceed this limit when attempting to do involved computations at compile time.
- *Maximum number of subexpressions evaluated within a single constant expression*: The suggested value for this limit as well as the default value for most implementations is 1,048,576, but it is important to note that this value can depend in surprising ways on the way that the number of subexpressions is calculated by each individual compiler. Expressions that stay within the limit reliably with one compiler might be counted differently in the constant expression evaluator of a different compiler, resulting in nonportable code. Compilers generally refer to this limit as the number of **constexpr** steps and do not always support adjusting it.

Though these limits can usually be increased with compiler flags, a significant overhead is introduced in terms of managing build options that can hinder how easily usable a library intended to be portable will be. Small differences in terms of how each compiler might count these values also hinder the ability to write portable **constexpr** code.

ing-constexpr-functions

Difficulty implementing `constexpr` functions

Many algorithms are simple to express iteratively and/or implement efficiently using dynamic data structures outside of what is possible within a **constexpr** function. Naive (and even non-naive) implementations often exceed the often wide-ranging limits that various compilers put on **constexpr** evaluation. Consider this straightforward implementation of `isPrime`:

```
template <typename T>
constexpr bool isPrime(T input)
{
    if (input < 2) return false;                // too small
    if (input == 2) return true;                // is two
```

```
    if (input % 2 == 0) return false;          // is even
    for (T i = 3; i <= input / i; i += 2)      // odd numbers up to square root
    {
        if (input % i == 0) { return false; }  // found divisor?
    }
    return true;                               // no divisors, input is prime
}
```

This implementation is iterative and fails to meet the requirements for being a C++11 **constexpr** function. While meeting the relaxed requirements for being a C++14 **constexpr** function (see Section 2.2."**??**" on page **??**), it is likely to hit default compiler limits on execution steps when `input` approaches 2^{40}.

To make this **constexpr** `isPrime` function implementation valid for C++11, we might start by switching to a recursive implementation for the same algorithm:

```
template <typename T>
constexpr bool isPrimeHelper(T n, T i)
{
    return n % i                           // i is not a divisor.
        && (   (i > n/i)                   // i is not larger than sqrt(n).
            || isPrimeHelper(n, i + 2));  // tail recursion on next i
}

template <typename T>
constexpr bool isPrime(T input)
{
    return input < 2                  ? false  // too small
         : (input == 2 || input == 3) ? true   // 2 or 3
         : (input % 2 == 0)           ? false  // odd
         : isPrimeHelper(input, 3);            // Call recursive helper.
}
```

The recursive implementation above works correctly, albeit slowly, up to an input value of around 2^{19}, hitting recursion limits on **constexpr** evaluation on most compilers. With significant effort, we might conceivably be able to push the upper limit slightly higher (e.g., by prechecking more factors than just 2). More importantly, recursively checking every other number below the square root of the input for divisibility is so slow compared to better algorithms that this approach is fundamentally inferior to a runtime solution.[26]

A final approach to working around the **constexpr** recursion limit is to implement a **divide and conquer** algorithm when searching the space of possible factors. While this approach has the same algorithmic performance as the directly recursive implementation and executes a comparable number of steps, the maximum recursion depth it needs is logarithmic in terms of the input value and will stay within the general compiler limits on recursion depth:

```
template <typename T>
```

[26]The C++20 Standard adds `std::is_constant_evaluated()`, a tool to allow a function to branch to different implementations at compile time and run time, enabling the compile-time algorithm to be different from the runtime algorithm with the same API.

```
constexpr bool hasFactor(T n, T begin, T end)
    // Return true if the specified n has a factor in the
    // closed range [begin, end], and false otherwise.
{
    return (begin > end)     ? false             // empty range [begin, end]
         : (begin > n/begin) ? false             // begin > sqrt(n)
         : (begin == end)    ? (n % begin == 0)  // [begin, end] has one element.
         :   // Otherwise, split into two ranges and recurse.
             hasFactor(n, begin, begin + (end - begin) / 2) ||
             hasFactor(n, begin + 1 + (end - begin) / 2, end);
}

template <typename T>
constexpr bool isPrime(T input)
    // Return true if the specified input is prime.
{
    return input < 2                  ? false  // too small
         : (input == 2 || input == 3) ? true   // 2 or 3
         : (input % 2 == 0)           ? false  // odd
         : !hasFactor(input, static_cast<T>(3), input - 1);
}
```

This C++11 implementation will generally work up to the same limits as the iterative C++14 implementation in the example above.

committing-to-constexpr

Prematurely committing to `constexpr`

Declaring a function to be **constexpr** comes with significant collateral costs that, to some, might not be obvious. Marking an eligible function **constexpr** would seem like a sure way to get compile-time evaluation, when possible (i.e., when constant expressions are passed into a function as parameters), without any additional cost for functions that currently meet the requirements of a **constexpr** function — essentially giving us a "free" runtime performance boost. The often-overlooked downside, however, is that this choice, once made, is not easily reversed. After a library is released and a **constexpr** function is evaluated as part of a constant expression, no clean way of turning back is available because clients now depend on this compile-time property.

overzealous-use-hurts

Overzealous use hurts

Overzealous application of **constexpr** can also have a significant impact on compilation time. Compile-time calculations can easily add seconds — or in extreme cases much more — to the compilation time of any translation unit that needs to evaluate them. When placed in a header file, these calculations need to be performed for all translation units that include that header file, vastly increasing total compilation time and hindering developer productivity. Compile-time precomputation *might* improve runtime performance in some cases but comes with other kinds of cost, which can be formidable.

Similarly, making public APIs that are **constexpr** usable without making it clear that they are suboptimal implementations can lead to both (1) excessive runtime overhead compared to a highly optimized non**constexpr** implementation (e.g., for `isPrime` in Difficulty

implementing **constexpr** functions) that might already exist in an organization's libraries and (2) increased compile time wherever algorithmically complex **constexpr** functions are invoked.

Compilation limits on compile-time evaluation are typically per *constant expression* and can easily be compounded unreasonably within just a single translation unit through the evaluation of numerous constant expressions. For example, when using the `generateArray` function in *Potential Pitfalls* — **??** on page **??**, [**AUs, there is no subsection called "Moving runtime calculation overhead to compile time." What did you mean?**] compile-time limits apply to each individual array element's computation, allowing total compilation to grow linearly with the number of values requested.

pile-time-or-run-time

One time is cheaper than compile time or run time

Overall, the ability to use a **constexpr** function to do calculations before run time fills in a spectrum of possibilities for who pays for certain calculations and when they pay for them, both in terms of computing time and maintenance costs.

Consider a possible set of evolutionary steps for a computationally expensive function that produces output values for a moderate number of unique input values. Examples include returning the timestamp for the start of a calendar year or returning the nth prime number up to some maximum n.

1. An initial version directly computes the output value each time it is needed. While correct and written entirely in maintainable C++, this version has the highest runtime overhead. Heavy use will quickly lead the developer to explore optimizations.

2. Where precomputing values might seem beneficial, a subsequent version initializes an array once at run time to avoid the extra computations. Aggregate runtime performance can be greatly improved but at the cost of slightly more code as well as a possibly noteworthy amount of runtime startup overhead. This hit at startup or on first use of the library can quickly become the next performance bottleneck that needs tackling. Initialization at startup can become increasingly problematic when linking large applications with a multitude of libraries, each of which might have moderate initialization times.

3. At this point **constexpr** comes into play as a tool to develop an option that avoids as much runtime overhead as possible. An initial such implementation puts the initialization of a **constexpr** array of values into the corresponding **inline** implementation in a library header. While this option minimizes the runtime overhead, the compile-time overhead now becomes significantly larger for every translation unit that depends on this library.

4. When faced with crippling compile times, the likely next step is to **insulate** the compile-time-generated table in an implementation file and to provide runtime access to it through accessor functions. While this removes the compilation overhead from clients who consume a binary distribution of the library, anyone who needs to build the library is still paying this cost each time they do a clean build. In modern environments, with widely disparate operating systems and build toolchains, source distributions have

become much more common and this overhead is imposed on a wide range of clients for a popular library.

5. Finally, the data table generation is moved into a separate program, often written in Python or some other non-C++ language. The output of this outboard program is then embedded as raw data (e.g., a sequence of numbers initializing an array) in the C++ implementation file. This solution eliminates the compile-time overhead for the C++ program; the cost of computing the table is paid only once by the developer. On the one hand, this solution adds to the maintenance costs for the initial developer, since a separate toolchain is often needed. On the other hand, the code becomes simpler, since the programmer is free to choose the best language for the job and is free from the constraints of **constexpr** in C++.

Thus, as attractive as it might seem to be able to precompute values directly in compile-time C++, complex situations often dictate against that choice. Note that a programmer with this knowledge might skip all of the intermediate steps and jump straight to the last one. For example, a list of prime numbers is readily available on the Internet without needing even to write a script; a programmer need only cut and paste it once, knowing that it will *never* change.

Annoyances

annoyances

Penalizing run time to enable compile time

-to-enable-compile-time

When adopting **constexpr** functions, programmers commonly forget that these functions are also called at run time, often more frequently than at compile time. Restrictions on the operations that are supported in a **constexpr** function definition, especially prior to the looser restrictions of C++14, will often lead to correct results that are less than optimally computed when executed at run time. A good example would be a **constexpr** implementation of the C function `strcmp`. Writing a recursive **constexpr** function to walk through two strings and return a result for the first characters that differ is relatively easy. However, most common implementations of this function are highly optimized, often taking advantage of inline assembly using architecture-specific vector instructions to handle multiple characters per CPU clock cycle. All of that fine tuning is given up if we rewrite the function to be **constexpr** compatible. Worse yet, the recursive nature of such functions prior to C++14 leads to a much greater risk of exceeding the limits of the stack, leading to program corruption and security risks when comparing long strings.

One possible workaround for these restrictions is to create different versions of the same function: a **constexpr** version usable at compile time and a non**constexpr** version optimized for run time. Since the language does not support overloading on **constexpr**, the end result is intrusive, requiring the different implementations to have different names. This complication can be mitigated by having a coding convention, such as placing all **constexpr** overloads in a namespace, say, `cexpr`, or giving all such functions the `_c` suffix. If the regular version of the function is also **constexpr**, the marked overload can simply forward all arguments to the regular function to ease maintenance, but overhead and complexity still come from having users manage multiple versions of the same function. It is not clear in all cases that the extra complexity covers its cost.

The relaxed restrictions in C++14 for implementing the bodies of **constexpr** functions is a welcome relief when optimizing for compile time and run time simultaneously; see Section 2.2."**??**" on page **??**. Even then, though, many runtime performance improvements (e.g., dynamic memory allocation, stateful caching, hardware intrinsics) are still not available to functions that need to execute at both run time and compile time. Note that a language-based solution that avoids the need to create separately named **constexpr** and non**constexpr** functions is introduced in C++20 with the `std::is_constant_evaluated()` intrinsic library function.

ualified-(c++11-only)

`constexpr` member functions are implicitly `const`-qualified (C++11 only)

A design flaw in C++11 (corrected in C++14) is that any member function declared **constexpr** is, where applicable, implicitly **const**-qualified, leading to unexpected behavior for member functions intended for use in a non**constexpr** context; see Section 2.2."**??**" on page **??**. This surprising restriction impacts code portability between language standards and makes the naive approach of just marking all member functions **constexpr** into something that unwittingly breaks what would otherwise be working functions.

see-also

See Also

- "**??**" (§2.1, p. **??**) ♦ is the companion use of the **constexpr** keyword applied to variables.
- "**??**" (§2.1, p. **??**) ♦ are often needed for complex metaprogramming used in some compile-time computations.
- "**??**" (§2.2, p. **??**) ♦ enumerates the significantly richer syntax permitted for implementing **constexpr** function bodies in C++14.

further-reading

Further Reading

None so far.

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

sec-unsafe-cpp11 Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14