5.8 — Constexpr and consteval functions

ALEX JANUARY 19, 2024

In lesson 5.5 -- Constexpr variables (https://www.learncpp.com/cpp-tutorial/constexpr-variables/), we introduced the `constexpr` keyword, which we used to create compile-time (symbolic) constants. We also introduced constant expressions, which are expressions that can be evaluated at compile-time rather than runtime.

Consider the following program, which uses two constexpr variables:

```
#include <iostream>

int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    std::cout << (x > y ? x : y) << " is greater!\n";

    return 0;
}
```

This produces the result:

```
6 is greater!
```

Because `x` and `y` are constexpr, the compiler can evaluate the constant expression `(x > y ? x : y)` at compile-time, reducing it to just `6`. Because this expression no longer needs to be evaluated at runtime, our program will run faster.

However, having a non-trivial expression in the middle of our print statement isn't ideal -- it would be better if the expression were a named function. Here's the same example using a function:

```
#include <iostream>

int greater(int x, int y)
{
    return (x > y ? x : y); // here's our expression
}

int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    std::cout << greater(x, y) << " is greater!\n"; // will be evaluated at runtime

    return 0;
}
```

This program produces the same output as the prior one. But there's a downside to putting our expression in a function: the call to `greater(x, y)` will execute at runtime. By using a function (which is good for modularity and documentation) we've lost our ability for that code to be evaluated at compile-time (which is bad for performance).

So how might we address this?

Constexpr functions can be evaluated at compile-time

A **constexpr function** is a function whose return value may be computed at compile-time. To make a function a constexpr function, we simply use the `constexpr` keyword in front of the return type. Here's a similar program to the one above, using a constexpr function:

```
#include <iostream>

constexpr int greater(int x, int y) // now a constexpr function
{
    return (x > y ? x : y);
}

int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    // We'll explain why we use variable g here later in the lesson
    constexpr int g { greater(x, y) }; // will be evaluated at compile-time

    std::cout << g << " is greater!\n";

    return 0;
}
```

This produces the same output as the prior example, but the function call `greater(x, y)` will be evaluated at compile-time instead of runtime!

When a function call is evaluated at compile-time, the compiler will calculate the return value of the function call, and then replace the function call with the return value.

So in our example, the call to `greater(x, y)` will be replaced by the result of the function call, which is the integer value `6`. In other words, the compiler will compile this:

```
#include <iostream>

int main()
{
    constexpr int x{ 5 };
    constexpr int y{ 6 };

    constexpr int g { 6 }; // greater(x, y) evaluated and replaced with return value 6

    std::cout << g << " is greater!\n";

    return 0;
}
```

To be eligible for compile-time evaluation, a function must have a constexpr return type and not call any non-constexpr functions when evaluated at compile-time. Additionally, a call to the function must have arguments that are constant expressions (e.g. compile-time constant variables or literals).

Author's note

We'll use the term "eligible for compile-time evaluation" later in the article, so remember this definition.

For advanced readers

There are some other lesser encountered criteria as well. These can be found here (https://en.cppreference.com/w/cpp/language/constexpr).

Our `greater()` function definition and function call in the above example meets these requirements, so it is eligible for compile-time evaluation.

Constexpr functions can also be evaluated at runtime

Functions with a constexpr return value can also be evaluated at runtime, in which case they will return a non-constexpr result. For example:


```
#include <iostream>

constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    int x{ 5 }; // not constexpr
    int y{ 6 }; // not constexpr

    std::cout << greater(x, y) << " is greater!\n"; // will be evaluated at runtime

    return 0;
}
```

In this example, because arguments `x` and `y` are not constant expressions, the function cannot be resolved at compile-time. However, the function will still be resolved at runtime, returning the expected value as a non-constexpr `int`.

Key insight

Allowing functions with a constexpr return type to be evaluated at either compile-time or runtime was allowed so that a single function can serve both cases.

Otherwise, you'd need to have separate functions (a function with a constexpr return type, and a function with a non-constexpr return type). This would not only require duplicate code, the two functions would also need to have different names!

This is also why C++ does not allow constexpr function parameters. A constexpr function parameter would imply the function could only be called with a constexpr argument. But this is not the case -- constexpr functions can be called with non-constexpr arguments when the function is evaluated at runtime.

So when is a constexpr function evaluated at compile-time?

You might think that a constexpr function would evaluate at compile-time whenever possible, but unfortunately this is not the case.

According to the C++ standard, a constexpr function that is eligible for compile-time evaluation must be evaluated at compile-time if the return value is used where a constant expression is required. Otherwise, the compiler is free to evaluate the function at either compile-time or runtime.

Let's examine a few cases to explore this further:

```
#include <iostream>

constexpr int greater(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    constexpr int g { greater(5, 6) };              // case 1: always evaluated at compile-time
    std::cout << g << " is greater!\n";

    std::cout << greater(5, 6) << " is greater!\n"; // case 2: may be evaluated at either runtime or compile-time

    int x{ 5 }; // not constexpr but value is known at compile-time
    std::cout << greater(x, 6) << " is greater!\n"; // case 3: likely evaluated at runtime

    std::cin >> x;
    std::cout << greater(x, 6) << " is greater!\n"; // case 4: always evaluated at runtime


    return 0;
}
```

In case 1, we're calling `greater()` with constant expression arguments, so it is eligible to be evaluated at compile-time. The initializer of constexpr variable `g` must be a constant expression, so the return value is used in a context that requires a constant expression. Thus, `greater()` must be evaluated at compile-time.

In case 2, the `greater()` function is again being called with constant expression arguments, so it is eligible for compile-time evaluation. However, the return value is not being used in a context that requires a constant expression (operator<< always executes at runtime), so the compiler is free to choose whether this call to `greater()` will be evaluated at compile-time or runtime!

In case 3, we're calling `greater()` with one argument that is not a constant expression. However, this argument has a value that is known at compile-time. Under the as-if rule, the compiler could decide to treat `x` as constexpr, and call evaluate this call to `greater()` at compile-time. But more likely, it will evaluate it at runtime.

In case 4, the value of argument `x` can't be known at compile-time, so this call to `greater()` will always evaluate at runtime.

Note that your compiler's optimization level setting may have an impact on whether it decides to evaluate a function at compile-time or runtime. This also means that your compiler may make different choices for debug vs. release builds (as debug builds typically have optimizations turned off).

Key insight

A constexpr function must be evaluated at compile-time if the return value is used where a constant expression is required. Otherwise,

compile-time evaluation is not guaranteed.

Thus, a constexpr function is better thought of as “can be used in a constant expression”, not “will be evaluated at compile-time”.

Key insight

Put another way, we can categorize the likelihood that a constexpr function will actually be evaluated at compile-time as follows:

Always (required by the standard):

- Constexpr function is called in context where constant expression is required.

Probably (there’s little reason not to):

- Constexpr function is called in context where constant expression isn’t required, all arguments are constant expressions.

Possibly (if optimized under the as-if rule):

- Constexpr function is called in context where constant expression isn’t required, some arguments are not constant expressions but their values are known at compile-time.

Never (not possible):

- Constexpr function is called in context where constant expression isn’t required, some arguments are not constant expressions and their values are not known at compile-time.

Determining if a constexpr function call is evaluating at compile-time or runtime

Prior to C++20, there are no standard language tools available to do this.

In C++20, `std::is_constant_evaluated()` (defined in the <type_traits> header) returns a `bool` indicating whether the current function call is executing in a constant context. This can be combined with a conditional statement to allow a function to behave differently when evaluated at compile-time vs runtime.

```
#include <type_traits> // for std::is_constant_evaluated

constexpr int someFunction()
{
    if (std::is_constant_evaluated()) // if compile-time evaluation
        // do something
    else // runtime evaluation
        // do something else
}
```

Used cleverly, you can have your function produce some observable difference (such as returning a special value) when evaluated at compile-time, and then infer how it evaluated from that result.

Forcing a constexpr function to be evaluated at compile-time

There is no way to tell the compiler that a constexpr function should prefer to evaluate at compile-time whenever it can (even in cases where the return value is used in a non-constant expression).

However, we can force a constexpr function that is eligible to be evaluated at compile-time to actually evaluate at compile-time by ensuring the return value is used where a constant expression is required. This needs to be done on a per-call basis.

The most common way to do this is to use the return value to initialize a constexpr variable (this is why we’ve been using variable ‘g’ in prior examples). Unfortunately, this requires introducing a new variable into our program just to ensure compile-time evaluation, which is ugly and

reduces code readability.

For advanced readers

There are several hacky ways that people have tried to work around the problem of having to introduce a new constexpr variable each time we want to force compile-time evaluation. See here (https://quuxplusone.github.io/blog/2018/08/07/force-constexpr/) and here (https://artificial-mind.net/blog/2020/11/14/cpp17-consteval).

However, in C++20, there is a better workaround to this issue, which we'll present in a moment.

Consteval C++20

C++20 introduces the keyword **consteval**, which is used to indicate that a function must evaluate at compile-time, otherwise a compile error will result. Such functions are called **immediate functions**.

```
#include <iostream>

consteval int greater(int x, int y) // function is now consteval
{
    return (x > y ? x : y);
}

int main()
{
    constexpr int g { greater(5, 6) };              // ok: will evaluate at compile-time
    std::cout << g << '\n';

    std::cout << greater(5, 6) << " is greater!\n"; // ok: will evaluate at compile-time

    int x{ 5 }; // not constexpr
    std::cout << greater(x, 6) << " is greater!\n"; // error: consteval functions must evaluate at compile-time

    return 0;
}
```

In the above example, the first two calls to `greater()` will evaluate at compile-time. The call to `greater(x, 6)` cannot be evaluated at compile-time, so a compile error will result.

Best practice

Use `consteval` if you have a function that must run at compile-time for some reason (e.g. performance).

Using consteval to make constexpr execute at compile-time C++20

The downside of consteval functions is that such functions can't evaluate at runtime, making them less flexible than constexpr functions, which can do either. Therefore, it would still be useful to have a convenient way to force constexpr functions to evaluate at compile-time (even when the return value is being used where a constant expression is not required), so that we could have compile-time evaluation when possible, and runtime evaluation when we can't.

Consteval functions provides a way to make this happen, using a neat helper function:

```
#include <iostream>

// Uses abbreviated function template (C++20) and `auto` return type to make this function work with any type of value
// See 'related content' box below for more info (you don't need to know how these work to use this function)
consteval auto compileTime(auto value)
{
    return value;
}

constexpr int greater(int x, int y) // function is constexpr
{
    return (x > y ? x : y);
}

int main()
{
    std::cout << greater(5, 6) << '\n';              // may or may not execute at compile-time
    std::cout << compileTime(greater(5, 6)) << '\n'; // will execute at compile-time

    int x { 5 };
    std::cout << greater(x, 6) << '\n';              // we can still call the constexpr version at runtime if we wish

    return 0;
}
```

This works because consteval functions require constant expressions as arguments -- therefore, if we use the return value of a constexpr function as an argument to a consteval function, the constexpr function must be evaluated at compile-time! The consteval function just returns this argument as its own return value, so the caller can still use it.

Note that the consteval function returns by value. While this might be inefficient to do at runtime (if the value was some type that is expensive to copy, e.g. std::string), in a compile-time context, it doesn't matter because the entire call to the consteval function will simply be replaced with the calculated return value.

Related content

We cover `auto` return types in lesson 10.9 -- Type deduction for functions (https://www.learncpp.com/cpp-tutorial/type-deduction-for-functions/).
We cover abbreviated function templates (`auto` parameters) in lesson 11.8 -- Function templates with multiple template types (https://www.learncpp.com/cpp-tutorial/function-templates-with-multiple-template-types/).

Constexpr/consteval functions are implicitly inline

Because constexpr functions may be evaluated at compile-time, the compiler must be able to see the full definition of the constexpr function at all points where the function is called. A forward declaration will not suffice, even if the actual function definition appears later in the same compilation unit.

This means that a constexpr function called in multiple files needs to have its definition included into each such file -- which would normally be a violation of the one-definition rule. To avoid such problems, constexpr functions are implicitly inline, which makes them exempt from the one-definition rule.

As a result, constexpr functions are often defined in header files, so they can be #included into any .cpp file that requires the full definition.

Consteval functions are also implicitly inline (presumably for consistency).

Rule

The compiler must be able to see the full definition of a constexpr or consteval function, not just a forward declaration.

Best practice

Constexpr/consteval functions used in a single source file (.cpp) can be defined in the source file above where they are used.

Constexpr/consteval functions used in multiple source files should be defined in a header file so they can be included into each source file.

Constexpr/consteval function parameters are not constexpr (but can be used as arguments to other constexpr functions)

The parameters of a constexpr function are not constexpr (and thus cannot be used in constant expressions). Such parameters can be declared as `const` (in which case they are treated as runtime constants), but not `constexpr`. This is because a constexpr function can be evaluated at runtime (which wouldn't be possible if the parameters were compile-time constants).

However, an exception is made in one case: a constexpr function can pass those parameters as arguments to another constexpr function, and that subsequent constexpr function can be resolved at compile-time. This allows constexpr functions to still be resolved at compile-time when

they call other constexpr functions (including themselves recursively).

Perhaps surprisingly, the parameters of a consteval function are not considered to be constexpr within the function either (even though consteval functions can only be evaluated at compile-time). This decision was made for the sake of consistency.

Here's a contrived example illustrating this:

```
#include <iostream>

constexpr int goo(int c)
{
    return c;
}

constexpr int foo(int b)
{
    constexpr int b2 { b }; // compile error: b is not a constant expression within foo()

    return goo(b);          // okay: b can still be used as argument to constexpr function goo()
}

int main()
{
    constexpr int a { 5 };

    std::cout << foo(a); // okay: constant expression a can be used as argument to constexpr function foo()

    return 0;
}
```

Related content

If you need constexpr parameters to a function (e.g. to use somewhere that requires a constant expression), see 11.9 -- Non-type template parameters (https://www.learncpp.com/cpp-tutorial/non-type-template-parameters/).

Can a constexpr function call a non-constexpr function?

The answer is yes, but only when the constexpr function is being evaluated in a non-constant context. A non-constexpr function may not be called when a constexpr function is evaluating in a constant context (because then the constexpr function wouldn't be able to produce a compile-time constant value).

Calling a non-constexpr function is allowed so that a constexpr function can do something like this:

```
#include <type_traits> // for std::is_constant_evaluated

constexpr int someFunction()
{
    if (std::is_constant_evaluated()) // if compile-time evaluation
        return someConstexprFcn();    // calculate some value at compile time
    else                              // runtime evaluation
        return someNonConstexprFcn(); // calculate some value at runtime
}
```

Now consider this variant:

```
constexpr int someFunction(bool b)
{
    if (b)
        return someConstexprFcn();
    else
        return someNonConstexprFcn();
}
```

This is legal as long as `someFunction(false)` is never called in a constant expression.

The C++ standard says that a constexpr function must return a constexpr value for at least one set of arguments, otherwise it is technically ill-formed. Therefore, calling a non-constexpr function unconditionally in a constexpr function makes the constexpr function ill-formed. However, compilers are not required to generate errors or warnings for such cases -- therefore, the compiler probably won't complain unless you try to call such a constexpr function in a constant context.

Therefore, we'd advise the following:

1. For best results, avoid calling non-constexpr functions from within a constexpr function if possible.
2. If your constexpr function requires different behavior for constant and non-constant contexts, conditionalize the behavior with `if (std::is_constant_evaluated())`.

3. Always test your constexpr functions in a constant context, as they may work when called in a non-constant context but fail in a constant context.

Why not constexpr every eligible function?

There are a few reasons you may not want to constexpr a function:

1. `constexpr` is part of the interface of a function. Once a function is made constexpr, it can be called by other constexpr functions or used in contexts that require constant expressions. Removing the `constexpr` later will break such code.
2. `constexpr` makes functions harder to debug because you can't inspect them at runtime.

But as a general rule, if you can `constexpr` a function, you should.

Best practice

Unless you have a specific reason not to, a function that can be made `constexpr` generally should be made `constexpr`.

Next lesson
5.9 Introduction to std::string

Back to table of contents

Previous lesson
5.7 Inline functions and variables

OKAY