5.5 — Constexpr variables

learncpp.com/cpp-tutorial/constexpr-variables/

In the previous lesson 5.4 -- Constant expressions and compile-time optimization, we defined what a constant expression is, as well as a compile-time constant. This lesson continues that discussion.

When you declare a const variable using the `const` keyword, the compiler will implicitly keep track of whether it's a runtime or compile-time constant. In most cases, this doesn't matter for anything other than optimization purposes, but there are a few cases where C++ requires a constant expression. And only compile-time constant variables can be used in a constant expression.

Because compile-time constants also allow for better optimization (and have little downside), we typically want to use compile-time constants wherever possible.

When using `const`, our variables could end up as either a compile-time const or a runtime const, depending on whether the initializer is a constant expression or not. In some cases, it can be hard to tell whether a const variable is a compile-time constant (and thus usable in a constant expression) or a runtime constant (and thus not usable in a constant expression).

For example:

```
int a { 5 };       // not const at all
const int b { a }; // obviously a runtime const (since initializer is non-const)
const int c { 5 }; // obviously a compile-time const (since initializer is a constant
expression)

const int d { obj };        // not obvious whether this is a runtime or compile-time
const
const int e { getValue() }; // not obvious whether this is a runtime or compile-time
const
```

In the above example, both `d` and `e` could be either a runtime or a compile-time const depending on how `obj` and `getValue()` are defined. It's not clear until we hunt down the definitions for those identifiers.

The `constexpr` keyword

Fortunately, we can enlist the compiler's help to ensure we get a compile-time constant variable where we desire one. To do so, we use the `constexpr` keyword instead of `const` in a variable's declaration. A **constexpr** (which is short for "constant expression") variable must be initialized with a constant expression, otherwise a compilation error will result.

For example:

```
#include <iostream>

int five()
{
    return 5;
}

int main()
{
    constexpr double gravity { 9.8 }; // ok: 9.8 is a constant expression
    constexpr int sum { 4 + 5 };      // ok: 4 + 5 is a constant expression
    constexpr int something { sum };  // ok: sum is a constant expression

    std::cout << "Enter your age: ";
    int age{};
    std::cin >> age;

    constexpr int myAge { age };      // compile error: age is not a constant
expression
    constexpr int f { five() };       // compile error: return value of five() is not
a constant expression

    return 0;
}
```

Note that non-integral types can be made constexpr.

Nomenclature

Because “constexpr” is short for “constant expression”, informally the term “constexpr” is often used as shorthand for “constant expression”. For example, we’d say the expression `1 + 2` is constexpr because the expression can be evaluated at compile-time.

Const vs constexpr

For variables, const means that the value of an object cannot be changed after initialization. Constexpr means that an object must have a value that is known at compile-time.

Constexpr variables are implicitly const. Const variables are not implicitly constexpr (except for const integral variables with a constant expression initializer).

Although a variable can be defined as both `constexpr` and `const`, in most cases this is redundant, and we only need to use either const or constexpr.

Best practice

Any constant variable whose initializer is a constant expression should be declared as `constexpr`.

Any constant variable whose initializer is not a constant expression should be declared as `const`.

Caveat: In the future we will discuss some types that are not fully compatible with `constexpr` (including `std::string`, `std::vector`, and other types that use dynamic memory allocation). For constant objects of these types, either use `const` instead of `constexpr`, or pick a different type that is constexpr compatible (e.g. `std::string_view` or `std::array`).

Author's note

Some of the examples on this site were written prior to the best practice to use constexpr -- as a result, you will note that many examples do not follow the above best practice. We are currently in the process of updating non-compliant examples as we run across them.

Const and constexpr function parameters

Normal function calls are evaluated at runtime, with the supplied arguments being used to initialize the function's parameters. Because the initialization of function parameters happens at runtime, this leads to two consequences:

1. `const` function parameters are treated as runtime constants (even when the supplied argument is a compile-time constant).
2. Function parameters cannot be declared as `constexpr`, since their initialization value isn't determined until runtime.

Related content

C++ does support functions that can be evaluated at compile-time (and thus can be used in constant expressions). We discuss these in lesson 5.8 -- Constexpr and consteval functions.

C++ also supports a way to pass compile-time constants to a function. We discuss these in lesson 11.9 -- Non-type template parameters.

Nomenclature recap

Term	**Definition**
Compile-time constant	An object whose value must be known at compile time (e.g. literals and constexpr variables).

Constexpr	Keyword that declares variables as compile-time constants (and functions that can be evaluated at compile-time). Informally, shorthand for “constant expression”.
Constant expression	An expression that contains only compile-time constants and operators/functions that support compile-time evaluation.
Runtime expression	An expression that is not a constant expression.
Runtime constant	An constant object that is not a compile-time constant.