

## 4.14 — Compile-time constants, constant expressions, and constexpr

1 ALEX 10 OCTOBER 3, 2022

Consider the following short program:

```
1 #include <iostream>

int main()
{
    int x { 3 + 4 };
    std::cout << x <<
    '\n';

    return 0;
}
```

The output is straightforward:

7

However, there's an interesting optimization possibility hidden within.

If this program were compiled exactly as it was written (with no optimizations), the compiler would generate an executable that calculates the result of `3 + 4` at runtime (when the program is run). If the program were executed a million times, `3 + 4` would be evaluated a million times, and the resulting value of `7` produced a million times. But note that the result of `3 + 4` never changes -- it is always `7`. So re-evaluating `3 + 4` every time the program is run is wasteful.

### Constant expressions

A **constant expression** is an expression that can be evaluated by the compiler at compile-time. To be a constant expression, all the values in the expression must be known at compile-time (and all of the operators and functions called must support compile-time evaluation).

When the compiler encounters a constant expression, it may evaluate the expression at compile-time, and then replace the constant expression with the result of the evaluation.

In the above program, the expression `3 + 4` is a constant expression. So when this program is compiled, the compiler may evaluate constant expression `3 + 4` and then replace the constant expression `3 + 4` with the resulting value `7`. In other words, the compiler may actually compile this:

```
1 #include <iostream>

int main()
{
    int x { 7 };
    std::cout << x <<
    '\n';

    return 0;
}
```

This program produces the same output (`7`), but the resulting executable no longer needs to spend CPU cycles calculating `3 + 4` at runtime!

Note that the expression `std::cout << x` is not a constant expression, because our program can't output values to the console at compile-time. So this expression will always evaluate at runtime.

### Key insight

Evaluating constant expressions at compile-time makes our compilation take longer (because the compiler has to do more work), but such expressions only need to be evaluated once (rather than every time the program is run). The resulting executables are faster and use less memory.

### As an aside...

The compiler is only required to evaluate constant expressions at compile time in contexts where a value is actually required at compile-time.

In the variable declaration `int x { 3 + 4 };`, `x` is not a constant variable and the initialization value does not need to be known at compile-time, so the constant expression `3 + 4` is not required to be evaluated at compile-time.

Even though it is not strictly required, modern compilers will usually evaluate a constant expression at compile-time because it is an easy optimization and more performant to do so.

## Compile-time constants

A **Compile-time constant** is a constant whose value is known at compile-time. Literals (e.g. `'1'`, `'2.3'`, and `"Hello, world!"`) are one type of compile-time constant.

But what about const variables? Const variables may or may not be compile-time constants.

## Compile-time const

A const variable is a compile-time constant if its initializer is a constant expression.

Consider a program similar to the above that uses const variables:

```
1 #include <iostream>

int main()
{
    const int x { 3 }; // x is a compile-time const
    const int y { 4 }; // y is a compile-time const

    const int z { x + y }; // x + y is a compile-time
    expression

    std::cout << z << '\n';

    return 0;
}
```

Because the initialization values of `x` and `y` are constant expressions, `x` and `y` are compile-time constants. This means `x + y` is also constant expression. So when the compiler compiles this program, it can evaluate `x + y` for their values, and replace the constant expression with the resulting literal `7`.

Note that the initializer of a compile-time const can be any constant expression. Both of the following will be compile-time const variables:

```
1 | const int a { 1 + 2 };  
   |  
   | const int b { z * 2 };
```

Compile-time const variables are often used as symbolic constants:

```
1 | const double gravity { 9.8 };
```

Compile-time constants enable the compiler to perform optimizations that aren't available with non-compile-time constants. For example, whenever `gravity` is used, the compiler can simply substitute the identifier `gravity` with the literal double `9.8`, which avoids having to fetch the value from somewhere in memory.

In many cases, compile-time constants will be optimized out of the program entirely. In cases where this is not possible (or when optimizations are turned off), the variable will still be created (and initialized) at runtime.

## Runtime const

Any const variable that is initialized with a non-constant expression is a runtime constant. **Runtime constants** are constants whose initialization values aren't known until runtime.

The following example illustrates the use of a constant that is a runtime constant:

```
1 | #include <iostream>  
  
   | int getNumber()  
   | {  
   |     std::cout << "Enter a number: ";  
   |     int y{};  
   |     std::cin >> y;  
   |  
   |     return y;  
   | }  
  
   | int main()  
   | {  
   |     const int x{ 3 };           // x is a compile time constant  
   |  
   |     const int y{ getNumber() }; // y is a runtime constant  
   |  
   |     const int z{ x + y };      // x + y is a runtime expression  
   |     std::cout << z << '\n';   // this is also a runtime  
   |     expression  
   |  
   |     return 0;  
   | }
```

Even though `y` is const, the initialization value (the return value of `getNumber()`) isn't known until runtime. Thus, `y` is a runtime constant, not a compile-time constant. And as such, the expression `x + y` is a runtime expression.

## The `constexpr` keyword

When you declare a `const` variable, 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 odd cases where C++ requires a compile-time constant instead of a run-time constant (we'll cover these cases later as we introduce those topics).

Because compile-time constants generally 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 compile-time expression or not. Because the definitions for both look identical, we can end up with a runtime const where we thought we were getting a compile-time const. In the previous example, it's hard to tell if `y` is a compile-time const or a runtime const -- we'd have to look at the return value of `getNumber()` to determine.

Fortunately, we can enlist the compiler's help to ensure we get a compile-time const where we expect 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 can only be a compile-time constant. If the initialization value of a `constexpr` variable is not a constant expression, the compiler will error.

For example:

```
1 #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
3   expression

       return 0;
   }
```

### Best practice

Any variable that should not be modifiable after initialization and whose initializer is known at compile-time should be declared as `constexpr`.

Any variable that should not be modifiable after initialization and whose initializer is not known at compile-time should be declared as `const`.

Although function parameters can be `const`, they cannot be `constexpr`.

### 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 [6.14 -- `constexpr` and `constexpr` functions](https://www.learncpp.com/cpp-tutorial/constexpr-and-constexpr-functions/) (<https://www.learncpp.com/cpp-tutorial/constexpr-and-constexpr-functions/>).

## Constant folding for constant subexpressions

Consider the following example:

```

1 | #include <iostream>
   |
   | int main()
   | {
   |     constexpr int x { 3 + 4 }; // 3 + 4 is a constant
   |     expression
   |     std::cout << x << '\n';    // this is a runtime expression
   |
   |     return 0;
   | }

```

`3 + 4` is a constant expression, so the compiler will evaluate `3 + 4` at compile-time, and replace it with value `7`. The compiler will likely optimize `x` out of the above program, replacing `std::cout << x << '\n'` with `std::cout << 7 << '\n'`. The output expression will execute at runtime.

However, because `x` is only used once, it's more likely we'd write the program like this in the first place:

```

1 | #include <iostream>
   |
   | int main()
   | {
   |     std::cout << 3 + 4 << '\n'; // this is a runtime
   |     expression
   |
   |     return 0;
   | }

```

Since the full expression `std::cout << 3 + 4 << '\n'` is not a constant expression, it's reasonable to wonder whether the constant subexpression `3 + 4` will still be optimized at compile-time. The answer is generally “yes”. Compilers have long been able to optimize constant subexpressions, including variables whose values can be determined at compile-time (compile-time const and constexpr variables).

### As an aside...

This optimization process is called “constant folding”.

Making our variables constexpr ensures that those variables have values known at compile-time, and thus are eligible for constant folding when they are used in expressions (even in non-const expressions).



**Next lesson**

4.15 [Literals](#)



**[Back to table of contents](#)**




**Previous lesson**



4.13 [Const variables and symbolic constants](#)

**B** **U** **URL** **INLINE CODE** **C++ CODE BLOCK** **HELP!**

Leave a comment...

 Name\*

 Email\* | 

 Find a mistake? Leave a comment! | 

Notify me about replies:



**POST COMMENT**

Avatars from <https://gravatar.com/> are connected to your provided email address.

**34 COMMENTS**

Newest ▾

