# 2.9 — Const, constexpr, and symbolic constants

BY ALEX ON FEBRUARY 23RD, 2015 | LAST MODIFIED BY ALEX ON JUNE 6TH, 2018

**Const variables**

So far, all of the variables we've seen have been non-constant -- that is, their values can be changed at any time. For example:

```
int x { 4 }; // initialize x with the value of 4
x = 5; // change value of x to 5
```

However, it's sometimes useful to define variables with values that can not be changed. For example, consider the value of gravity on Earth: 9.8 meters/second^2. This isn't likely to change any time soon. Defining this value as a constant helps ensure that this value isn't accidentally changed.

To make a variable constant, simply put the const keyword either before or after the variable type, like so:

```
const double gravity { 9.8 }; // preferred use of const before type
int const sidesInSquare { 4 }; // okay, but not preferred
```

Although C++ will accept const either before or after the type, we recommend using it before the type because it better follows standard English language convention where modifiers come before the object being modified (e.g. a green ball, not a ball green).

Const variables *must* be initialized when you define them, and then that value can not be changed via assignment.

Declaring a variable as const prevents us from inadvertently changing its value:

```
const double gravity { 9.8 };
gravity = 9.9; // not allowed, this will cause a compile error
```

Defining a const variable without initializing it will also cause a compile error:

```
const double gravity; // compiler error, must be initialized upon definition
```

Note that const variables can be initialized from non-const values:

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

const int usersAge (age); // usersAge can not be changed
```

Const is used most often with function parameters:

```
void printInteger(const int myValue)
{
    std::cout << myValue;
}
```

Making a function parameter const does two things. First, it tells the person calling the function that the function will not change the value of myValue. Second, it ensures that the function doesn't change the value of myValue.

With parameters passed by value, like the above, we generally don't care if the function changes the value of the parameter (since it's just a copy that will be destroyed at the end of the function anyway). For this reason, we usually don't const parameters passed by value. But later on, we'll talk about other kinds of function parameters (where changing the value of the parameter will change the value of the argument passed in). For these types of parameters, judicious use of const is important.

**Compile time vs runtime**

When you are in the process of compiling your program, that is called **compile time**. During compile time, the compiler ensures your code is syntactically correct, and converts your code into object files.

When you are in the process of running your application, that is called **runtime**. During runtime, your program executes line by line.

**Constexpr**

C++ actually has two different kinds of constants.

Runtime constants are those whose initialization values can only be resolved at runtime (when your program is running). Variables such as usersAge and myValue above are runtime constants, because the compiler can't determine their values at compile time. usersAge relies on user input (which can only be given at runtime) and myValue depends on the value passed into the function (which is only known at runtime).

Compile-time constants are those whose initialization values can be resolved at compile-time (when your program is compiling). Variable gravity above is an example of a compile-time constant. Whenever gravity is used, the compiler can simply substitute the identifier gravity for the literal double 9.8.

In most cases, it doesn't matter whether a constant value is runtime or compile-time. However, there are a few odd cases where C++ requires a compile-time constant instead of a run-time constant (such as when defining the length of a fixed-size array -- we'll cover this later). Because a const value could be either runtime or compile-time, the compiler has to keep track of which kind of constant it is.

To help provide more specificity, C++11 introduced new keyword **constexpr**, which ensures that the constant must be a compile-time constant:

```
constexpr double gravity (9.8); // ok, the value of 9.8 can be resolved at compile-time
constexpr int sum = 4 + 5; // ok, the value of 4 + 5 can be resolved at compile-time

std::cout << "Enter your age: ";
int age;
std::cin >> age;
constexpr int myAge = age; // not okay, age can not be resolved at compile-time
```

*Rule: Any variable that should not change values after initialization and whose initializer is known at compile-time should be declared as constexpr.*
*Rule: Any variable that should not change values after initialization and whose initializer is not known at compile-time should be declared as const.*

Note: Many of these tutorials were written before constexpr existed and have not been fully updated. In practical terms, this isn't an issue, as constexpr is only required in specific situations.

**Naming your const variables**

Some programmers prefer to use all upper-case names for const variables. Others use normal variable names with a 'k' prefix. However, we will use normal variable naming conventions, which is more common. Const variables act exactly like normal variables in every case except that they can not be assigned to, so there's no particular reason they need to be denoted as special.

**Symbolic constants**

In the previous lesson **2.8 -- Literals**, we discussed "magic numbers", which are literals used in a program to represent a constant value. Since magic numbers are bad, what should you do instead? The answer is: use symbolic constants! A **symbolic constant** is a name given to a constant literal value. There are two ways to declare symbolic constants in C++. One of them is good, and one of them is not. We'll show you both.

**Bad: Using object-like macros with a substitution parameter as symbolic constants**

We're going to show you the less desirable way to define a symbolic constant first. This method was commonly used in a lot of older code, so you may still see it.

In lesson **1.10 -- A first look at the preprocessor**, you learned that object-like macros have two forms -- one that doesn't take a substitution parameter (generally used for conditional compilation), and one that does have a substitution parameter. We'll talk about the case with the substitution parameter here. That takes the form:

```
#define identifier substitution_text
```

Whenever the preprocessor encounters this directive, any further occurrence of 'identifier' is replaced by 'substitution_text'. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

Consider the following snippet:

```
#define MAX_STUDENTS_PER_CLASS 30
```

```
int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
```

When you compile your code, the preprocessor replaces all instances of MAX_STUDENTS_PER_CLASS with the literal value 30, which is then compiled into your executable.

You'll likely agree that this is much more intuitive than using a magic number for a couple of reasons. MAX_STUDENTS_PER_CLASS provides context for what the program is trying to do, even without a comment. Second, if the number of max students per classroom changes, we only need to change the value of MAX_STUDENTS_PER_CLASS in one place, and all instances of MAX_STUDENTS_PER_CLASS will be replaced by the new literal value at the next compilation.

Consider our second example, using #define symbolic constants:

```
#define MAX_STUDENTS_PER_CLASS 30
#define MAX_NAME_LENGTH 30

int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
setMax(MAX_NAME_LENGTH);
```

In this case, it's clear that MAX_STUDENTS_PER_CLASS and MAX_NAME_LENGTH are intended to be independent values, even though they happen to share the same value (30).

So why not use #define to make symbolic constants? There are (at least) two major problems.

First, because macros are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger (which shows you your actual code). So although the compiler would compile `int max_students = numClassrooms * 30;`, in the debugger you'd see `int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;`. You'd have to go find the definition of MAX_STUDENTS_PER_CLASS in order to know what the actual value was. This can make your programs harder to debug.

Second, #defined values always have file scope (which we'll talk more about in the section on local and global variables). This means a value #defined in one piece of code may have a naming conflict with a value #defined with the same file later.

For example:

```
#include <iostream>

void a()
{
// This define value is now available for the rest of this file
#define x 5
    std::cout << x;
}

void b()
{
// Even though we're intending this x to be local to function b()
// it conflicts with the x we defined inside function a()
#define x 6
    std::cout << x;
}

int main() {

    a();
    b();

    return 0;
}
```

In the above code, we #define x in function a(), intending it to be used inside function a(). However, #define value x can actually be used anywhere beyond that point in the same file. So when function b() #defines its own x, we get a naming conflict. Even if function b() used a variable named x instead of a #define value named x, we would still have issues, as the preprocessor would try to replace variable name x with the value 5.

*Rule: Avoid using #define to create symbolic constants*

**A better solution: Use const variables**

A better way to create symbolic constants is through use of const (or better, constexpr) variables:

```
constexpr int maxStudentsPerClass { 30 };
constexpr int maxNameLength { 30 };
```

These will show up in the debugger, and follow all of the normal variable rules around scope.

*Rule: use const variables to provide a name and context for your magic numbers.*

**Using symbolic constants throughout a program**

In many applications, a given symbolic constant needs to be used throughout your code (not just in one location). These can include physics or mathematical constants that don't change (e.g. pi or avogadro's number), or application-specific "tuning" values (e.g. friction or gravity coefficients). Instead of redefining these every time they are needed, it's better to declare them once in a central location and use them wherever needed. That way, if you ever need to change them, you only need to change them in one place.

There are multiple ways to facilitate this within C++, but the following is probably easiest:

1) Create a header file to hold these constants
2) Inside this header file, declare a namespace (we'll talk more about this in lesson **4.3b -- Namespaces**)
3) Add all your constants inside the namespace (make sure they're const)
4) #include the header file wherever you need it

e.g. constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace constants
{
    constexpr double pi(3.14159);
    constexpr double avogadro(6.0221413e23);
    constexpr double my_gravity(9.2); // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

Use the scope resolution operator (::) to access your constants in .cpp files:

```
#include "constants.h"
double circumference = 2 * radius * constants::pi;
```

If you have both physics constants and per-application tuning values, you may opt to use two sets of files -- one for the physics values that will never change, and one for your per-program tuning values that are specific to your program. That way you can reuse the physics values in any program.

Note: In lesson **4.2 -- Global variables and linkage**, we show a more efficient way to do symbolic constants using global variables.

**2.10 -- Chapter 2 comprehensive quiz**

**Index**

**2.8 -- Literals**