4.14 — Const, constexpr, and symbolic constants

ALEX JUNE 28, 2021

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 gravity of Earth (near the surface): 9.8 meters/second^2. This isn't likely to change any time soon (and if it does, you've likely got bigger problems than learning C++). 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 *const* 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

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

```cpp
int main()
{
    const double gravity; // error: const variables must be initialized
    gravity = 9.9; // error: const variables can not be changed

    return 0;
}
```

Note that const variables can be initialized from other variables (including non-const ones):

```cpp
#include <iostream>

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

    const int usersAge { age };

    // age is non-const and can be changed
    // usersAge is const and can not be changed

    return 0;
}
```

Const function parameters

Const can also be used with function parameters:

```cpp
#include <iostream>

void printInt(const int x)
{
    std::cout << x;
}

int main()
{
    printInt(5); // 5 will be used as the initializer for x
    printInt(6); // 6 will be used as the initializer for x

    return 0;
}
```

Making a function parameter const enlists the compiler's help to ensure that the parameter's value is not changed inside the function. Note that we did not provide an explicit initializer for our const parameter -- the value of the argument in the function call will be used as the initializer in this case.

When arguments are passed by value, 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 other types of parameters, use of const is important.

Runtime vs compile-time constants

C++ actually has two different kinds of constants.

Runtime constants are constants whose initialization values can only be resolved at runtime (when your program is running). The following are examples of runtime constants:

```cpp
#include <iostream>

void printInt(const int x) // x is a runtime constant because the value isn't known until the program is run
{
    std::cout << x;
}

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

    const int usersAge { age }; // usersAge is a runtime constant because the value isn't known until the program is run

    std::cout << "Your age is: ";
    printInt(age);

    return 0;
}
```

Variables such as *usersAge* and *x* in the above program above are runtime constants, because the compiler can't determine their initial values until the program is actually run. *usersAge* relies on user input (which can only be given at runtime) and *x* depends on the value passed into the function (which is only known at runtime). However, once initialized, the value of these constants can't be changed.

Compile-time constants are constants whose initialization values can be determined at compile-time (when your program is compiling). The following are examples of compile-time constants:

```cpp
const double gravity { 9.8 }; // the compiler knows at compile-time that gravity will have value 9.8
const int something { 1 + 2 }; // the compiler can resolve this at compiler time
```

Compile-time constants enable the compiler to perform optimizations that aren't available with runtime constants. For example, whenever *gravity* is used, the compiler can simply substitute the identifier *gravity* with the literal double *9.8*.

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, 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).

---

constexpr

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

```cpp
#include <iostream>

int main()
{
    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 }; // compile error: age is a runtime constant, not a compile-time constant

    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.

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 4.13 -- 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 2.9 -- Introduction to 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:

```cpp
#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). That way, if we need to update our classroom size, we won't accidentally change the name length too.

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

First, because macros are resolved by the preprocessor, all occurrences of the macro are replaced with the defined value just prior to compilation. If you are debugging your code, you won't see the actual value (e.g. `30`) -- you'll only see the name of the symbolic constant (e.g. `MAX_STUDENTS_PER_CLASS`). And because these #defined values aren't variables, you can't add a watch in the debugger to see their values. If you want to know what value `MAX_STUDENTS_PER_CLASS` resolves to, you'll have to find the definition of `MAX_STUDENTS_PER_CLASS` (which could be in a different file). This can make your programs harder to debug.

Second, macros can conflict with normal code. For example:

```
#include
"someheader.h"
#include <iostream>

int main()
{
    int beta { 5 };
    std::cout <<
beta;

    return 0;
}
```

If someheader.h happened to #define a macro named beta, this simple program would break, as the preprocessor would replace the int variable beta's name with whatever the macro's value was.

Thirdly, macros don't follow normal scoping rules, which means in rare cases a macro defined in one part of a program can conflict with code written in another part of the program that it wasn't supposed to interact with.

Warning

Avoid using #define to create symbolic constants macros.

A better solution: Use constexpr variables

A better way to create symbolic constants is through use of constexpr variables:

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

Because these are just normal variables, they are watchable in the debugger, have normal scoping, and avoid other weird behaviors.

Best practice

Use constexpr variables to provide a name and context for your magic numbers.

Using symbolic constants throughout a multi-file 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++ -- we cover this topic in full detail in lesson 6.9 -- Sharing global constants across multiple files (using inline variables).

Next lesson

4.x Chapter 4 summary and quiz

Back to table of contents

Previous lesson

4.13 Literals

DP N N FOUT

Newest