4.13 — Const variables and symbolic constants

ALEX AUGUST 2, 2022

In programming, a **constant** is a value that may not be changed. C++ supports several types of constants: const variables (which we'll cover in this lesson and 4.14 -- Compile-time constants, constant expressions, and constexpr (https://www.learncpp.com/cpp-tutorial/compile-time-constants-constant-expressions-and-constexpr/)), and literals (which we'll cover shortly, in lesson 4.15 -- Literals (https://www.learncpp.com/cpp-tutorial/literals/)).

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 (typically through assignment of a new value). For example:

```cpp
int main()
{
    int x { 4 }; // x is a non-constant variable
    x = 5; // change value of x to 5 using assignment
operator

    return 0;
}
```

However, there are many cases where it is 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. Constants also have other benefits that we'll explore momentarily.

A variable whose value can not be changed is called a **constant variable**.

The const keyword

To make a variable a constant, place the `const` keyword in the variable's declaration either before or after the variable type, like so:

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

Although C++ will accept `const` either before or after the type, it's much more common to use `const` before the type because it better follows standard English language convention where modifiers come before the object being modified (e.g. a “a green ball”, not a “a ball green”).

As an aside…

Due to the way that the compiler parses more complex declarations, some developers prefer placing the `const` after the type (because it is slightly more consistent). This style is called “east const”. While this style has some advocates (and some reasonable points), it has not caught on significantly.

Best practice

Place `const` before the type (because it is more idiomatic to do so).

Const variables must be initialized

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

```
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):

```
#include <iostream>

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

    const int constAge { age }; // initialize const variable using non-const value

    age = 5;      // ok: age is non-const, so we can change its value
    constAge = 6; // error: constAge is const, so we cannot change its value

    return 0;
}
```

In the above example, we initialize const variable `constAge` with non-const variable `age`. Because `age` is still non-const, we can change its value. However, because `constAge` is const, we cannot change the value it has after initialization.

Naming your const variables

There are a number of different naming conventions that are used for const variables.

Programmers who have transitioned from C often prefer underscored, upper-case names for const variables (e.g. `EARTH_GRAVITY`). More common in C++ is to use intercapped names with a ‘k’ prefix (e.g. `kEarthGravity`).

However, because const variables act like normal variables (except they can not be assigned to), there is no reason that they need a special naming convention. For this reason, we prefer using the same naming convention that we use for non-const variables (e.g. `earthGravity`).

Const function parameters

Function parameters can be made constants via the `const` keyword:

```
#include <iostream>

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

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;
}
```

Note that we did not provide an explicit initializer for our const parameter `x` -- the value of the argument in the function call will be used as the initializer for `x`.

Making a function parameter constant enlists the compiler's help to ensure that the parameter's value is not changed inside the function. However, 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 (as it adds clutter to our code without providing much actual value).

Best practice

Don't use `const` when passing by value.

Later in this tutorial series, we'll talk about two other ways to pass arguments to functions: pass by reference, and pass by address. When using either of these methods, proper use of `const` is important.

Const return values

A function's return value may also be made const:

```
#include <iostream>

const int getValue()
{
    return 5;
}

int main()
{
    std::cout << getValue() <<
'\n';

    return 0;
}
```

However, since the returned value is a copy, there's little point in making it `const`. Returning a const value can also impede certain kinds of compiler optimizations, which can result in lower performance.

Best practice

Don't use `const` when returning by value.

What is a symbolic constant?

A **symbolic constant** is a name that is given to a constant value. Constant variables are one type of symbolic constant, as a variable has a name (its identifier) and a constant value.

In lesson 2.10 -- Introduction to the preprocessor (https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/), we discussed that the preprocessor supports object-like macros with substitution text. These take the form:

```
#define identifier substitution_text
```

Whenever the preprocessor processes 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.

For example:

```
#include <iostream>
#define MAX_STUDENTS_PER_CLASS 30

int main()
{
    std::cout << "The class has " << MAX_STUDENTS_PER_CLASS << "
students.\n";

    return 0;
}
```

When compiling this program, the preprocessor will replace `MAX_STUDENTS_PER_CLASS` with the literal value `30`, which the compiler will then compile into your executable.

Because object-like macros have a name, and the substitution text is a constant value, object-like macros with substitution text are also symbolic constants.

For symbolic constants, prefer constant variables to object-like macros

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 have naming conflicts with normal code. For example:

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

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

    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 the macro's substitution text. This is normally avoided by using all caps for macro names, but it can still happen.

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.

Best practice

Prefer constant variables over object-like macros with substitution text.

Using constant variables 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) (https://www.learncpp.com/cpp-tutorial/sharing-global-constants-across-multiple-files-using-inline-variables/).

Next lesson
4.14 Compile-time constants, constant expressions, and constexpr

Back to table of contents

Previous lesson
4.12 Introduction to type conversion and static_cast