5.1 — Constant variables (named constants)

Introduction to constants

In programming, a **constant** is a value that may not be changed during the program's execution.

C++ supports two different kinds of constants:

- **Named constants** are constant values that are associated with an identifier. These are also sometimes called **symbolic constants**, or occasionally just **constants**.
- **Literal constants** are constant values that are not associated with an identifier.

We'll start our coverage of constants by looking at named constants. We will then cover literal constants (in upcoming lesson 5.2 -- Literals).

Types of named constants

There are three ways to define a named constant in C++:

- Constant variables (covered in this lesson).
- Object-like macros with substitution text (introduced in lesson 2.10 -- Introduction to the preprocessor, with additional coverage in this lesson).
- Enumerated constants (covered in lesson 13.2 -- Unscoped enumerations).

Constant variables are the most common type of named constant, so we'll start there.

Constant 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 by assigning a new value). For example:

```
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 in subsequent lessons.

Although it is a well-known oxymoron, a variable whose value cannot be changed is called a **constant variable**.

Declaring a const variable

To declare a constant variable, we place the `const` keyword (called a "const qualifier") adjacent to the object's type:

```
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 the const qualifier 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 conventional 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, in modern C++ we don't make value parameters `const` because 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). The `const` keyword also adds a small amount of unnecessary clutter to the function prototype.

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;
}
```

For fundamental types, the `const` qualifier on a return type is simply ignored (your compiler may generate a warning).

For other types (which we'll cover later), there is typically little point in returning const objects by value, because they are temporary copies that will be destroyed anyway. Returning a const value can also impede certain kinds of compiler optimizations (involving move semantics), which can result in lower performance.

Best practice

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

Object-like macros with substitution text

In lesson 2.10 -- Introduction to the preprocessor, we discussed object-like macros with substitution text. For example:

```
#include <iostream>

#define MY_NAME "Alex"

int main()
{
    std::cout << "My name is: " << MY_NAME << '\n';

    return 0;
}
```

When the preprocessor processes the file containing this code, it will replace `MY_NAME` (on line 7) with `"Alex"`. Note that `MY_NAME` is a name, and the substitution text is a constant value, so object-like macros with substitution text are also named constants.

Prefer constant variables to preprocessor macros

So why not use preprocessor macros for named constants? There are (at least) three major problems.

The biggest issue is that macros don't follow normal C++ scoping rules. Once a macro is #defined, all subsequent occurrences of the macro's name in the current file will be replaced. If that name is used elsewhere, you'll get macro substitution where you didn't want it. This will most likely lead to strange compilation errors. For example:

```
#include <iostream>

void someFcn()
{
// Even though gravity is defined inside this function
// the preprocessor will replace all subsequent occurrences of gravity in the rest of
the file
#define gravity 9.8
}

void printGravity(double gravity) // including this one, causing a compilation error
{
    std::cout << "gravity: " << gravity << '\n';
}

int main()
{
    printGravity(3.71);

    return 0;
}
```

When compiled, GCC produced this confusing error:

```
prog.cc:5:17: error: expected ',' or '...' before numeric constant
    5 | #define gravity 9.8
      |                 ^~~
prog.cc:9:26: note: in expansion of macro 'gravity'
```

Second, it is often harder to debug code using macros. Although your source code will have the macro's name, the compiler and debugger never see the macro because it has already been replaced before they run. Many debuggers are unable to inspect a macro's value, and often have limited capabilities when working with macros.

Third, macro substitution behaves differently than everything else in C++. Inadvertent mistakes can be easily made as a result.

Constant variables have none of these problems: they follow normal scoping rules, can be seen by the compiler and debugger, and behave consistently.

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 named constant needs to be used throughout your code (not just in one file). 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 7.9 -- Sharing global constants across multiple files (using inline variables).

Type qualifiers

A **type qualifier** (sometimes called a **qualifier** for short) is a keyword that is applied to a type that modifies how that type behaves. The `const` used to declare a constant variable is called a **const type qualifier** (or **const qualifier** for short).

As of C++23, C++ only has two type qualifiers: `const` and `volatile`.

Optional reading

The `volatile` qualifier is used to tell the compiler that an object may have its value changed at any time. This rarely-used qualifier disables certain types of optimizations.

In technical documentation, the `const` and `volatile` qualifiers are often referred to as **cv-qualifiers**. The following terms are also used in the C++ standard:

- A **cv-unqualified** type is a type with no type qualifiers (e.g. `int`).
- A **cv-qualified** type is a type with one or more type qualifiers applied (e.g. `const int`).
- A **possibly cv-qualified** type is a type that may be cv-unqualified or cv-qualified.

These terms are not used much outside of technical documentation, so they are listed here for reference, not as something you need to remember.