8.2 — If statements and blocks

The first category of control flow statements we'll talk about is conditional statements. A **conditional statement** is a statement that specifies whether some associated statement(s) should be executed or not.

C++ supports two basic kinds of conditionals: `if statements` (which we introduced in lesson 4.10 -- Introduction to if statements, and will talk about further here) and `switch statements` (which we'll cover in a couple of lessons).

Quick if-statement recap

The most basic kind of conditional statement in C++ is the `if statement`. An `if statement` takes the form:

```
if (condition)
    true_statement;
```

or with an optional `else statement`:

```
if (condition)
    true_statement;
else
    false_statement;
```

If the `condition` evaluates to `true`, the `true_statement` executes. If the `condition` evaluates to `false` and the optional `else statement` exists, the `false_statement` executes.

Here is a simple program that uses an `if statement` with the optional `else statement`:

```
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    int x{};
    std::cin >> x;

    if (x > 10)
        std::cout << x << " is greater than 10\n";
    else
        std::cout << x << " is not greater than 10\n";

    return 0;
}
```

This program works just like you'd expect:

```
Enter a number: 15
15 is greater than 10

Enter a number: 4
4 is not greater than 10
```

If or else with multiple conditional statements

New programmers often try something like this:

```
#include <iostream>

namespace constants
{
    constexpr int minRideHeightCM { 140 };
}

int main()
{
    std::cout << "Enter your height (in cm): ";
    int x{};
    std::cin >> x;

    if (x >= constants::minRideHeightCM)
        std::cout << "You are tall enough to ride.\n";
    else
        std::cout << "You are not tall enough to ride.\n";
        std::cout << "Too bad!\n"; // focus on this line

    return 0;
}
```

However, consider the following run of the program:

```
Enter your height (in cm): 180
You are tall enough to ride.
Too bad!
```

This program doesn't work as expected because the `true_statement` and `false_statement` can only be a single statement. The indentation is deceiving us here -- the above program executes as if it had been written as follows:

```
#include <iostream>

namespace constants
{
    constexpr int minRideHeightCM { 140 };
}

int main()
{
    std::cout << "Enter your height (in cm): ";
    int x{};
    std::cin >> x;

    if (x >= constants::minRideHeightCM)
        std::cout << "You are tall enough to ride.\n";
    else
        std::cout << "You are not tall enough to ride.\n";

    std::cout << "Too bad!\n"; // focus on this line

    return 0;
}
```

This makes it clearer that “Too bad!” will always execute.

However, it’s common to want to execute multiple statements based on some condition. To do so, we can use a compound statement (block):

```
#include <iostream>

namespace constants
{
    constexpr int minRideHeightCM { 140 };
}

int main()
{
    std::cout << "Enter your height (in cm): ";
    int x{};
    std::cin >> x;

    if (x >= constants::minRideHeightCM)
        std::cout << "You are tall enough to ride.\n";
    else
    { // note addition of block here
        std::cout << "You are not tall enough to ride.\n";
        std::cout << "Too bad!\n";
    }

    return 0;
}
```

Remember that blocks are treated as a single statement, so this now works as expected:

```
Enter your height (in cm): 180
You are tall enough to ride.

Enter your height (in cm): 130
You are not tall enough to ride.
Too bad!
```

To block or not to block single statements

There is debate within the programmer community as to whether single statements following an `if` or `else` should be explicitly enclosed in blocks or not.

There are two reasons typically given as rationale for doing so. First, consider the following snippet:

```
if (age >= minDrinkingAge)
    purchaseBeer();
```

Now let's say we're in a hurry and modify this program to add another ability:

```
if (age >= minDrinkingAge)
    purchaseBeer();
    gamble(); // will always execute
```

Oops, we've just allowed minors to gamble. Have fun in jail!

Second, it can make programs more difficult to debug. Let's say we have the following snippet:

```
if (age >= minDrinkingAge)
    addBeerToCart();

checkout();
```

Let's say we suspect something is wrong with the `addBeerToCart()` function, so we comment it out:

```
if (age >= minDrinkingAge)
//    addBeerToCart();

checkout();
```

Now we've made `checkout()` conditional, which we certainly didn't intend.

Neither of these problems occur if you always use blocks after an `if` or `else` statement.

The best argument for not using blocks around single statements is that adding blocks makes you able to see less of your code at one time by spacing it out vertically, which makes your code less readable and can lead to other, more serious mistakes.

The community seems to be more in favor of always using blocks than not, though this recommendation certainly isn't ubiquitous.

Best practice

Consider putting single statements associated with an `if` or `else` in blocks (particularly while you are learning). More experienced C++ developers sometimes disregard this practice in favor of tighter vertical spacing.

A middle-ground alternative is to put single-lines on the same line as the `if` or `else`:

```
if (age >= minDrinkingAge) purchaseBeer();
```

This avoids both of the above downsides mentioned above at some minor cost to readability.

Implicit blocks

If the programmer does not declare a block in the statement portion of an `if statement` or `else statement`, the compiler will implicitly declare one. Thus:

```
if (condition)
    true_statement;
else
    false_statement;
```

is actually the equivalent of:

```
if (condition)
{
    true_statement;
}
else
{
    false_statement;
}
```

Most of the time, this doesn't matter. However, new programmers sometimes try to do something like this:

```
#include <iostream>

int main()
{
    if (true)
        int x{ 5 };
    else
        int x{ 6 };

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

    return 0;
}
```

This won't compile, with the compiler generating an error that identifier `x` isn't defined. This is because the above example is the equivalent of:

```
#include <iostream>

int main()
{
    if (true)
    {
        int x{ 5 };
    } // x destroyed here
    else
    {
        int x{ 6 };
    } // x destroyed here

    std::cout << x << '\n'; // x isn't in scope here

    return 0;
}
```

In this context, it's clearer that variable `x` has block scope and is destroyed at the end of the block. By the time we get to the `std::cout` line, `x` doesn't exist.

We'll continue our exploration of `if statements` in the next lesson.