# 5.2 — If statements

BY ALEX ON JUNE 21ST, 2007 | LAST MODIFIED BY ALEX ON APRIL 29TH, 2018

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

```
if (expression)
    statement
```

or

```
if (expression)
    statement
else
    statement2
```

The expression is called a **conditional expression**. If the expression evaluates to true (non-zero), the statement executes. If the expression evaluates to false, the *else statement* is executed if it exists.

Here is a simple program that uses an *if 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;
}
```

## Using if with multiple statements

Note that the *if statement* only executes a single statement if the expression is true, and the *else* only executes a single statement if the expression is false. In order to execute multiple statements, we can use a block:

```
#include <iostream>

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

    if (x > 10)
        {
        // both statements will be executed if x > 10
        std::cout << "You entered " << x << "\n";
        std::cout << x << "is greater than 10\n";
        }
    else
        {
        // both statements will be executed if x <= 10
        std::cout << "You entered " << x << "\n";
        std::cout << x << "is not greater than 10\n";
        }

    return 0;
}
```

**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 (expression)
    statement
else
    statement2
```

is actually the equivalent of:

```
if (expression)
{
    statement
}
else
{
    statement2
}
```

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

```
#include <iostream>

int main()
{
    if (1)
        int x = 5;
    else
        int x = 6;

    std::cout << x;

    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 (1)
    {
        int x = 5;
    } // x destroyed here
    else
    {
        int x = 6;
    } // x destroyed here

    std::cout << x; // x isn't defined 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.

**Chaining if statements**

It is possible to chain if-else statements together:

```cpp
#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 if (x < 10)
        std::cout << x << "is less than 10\n";
    else
        std::cout << x << "is exactly 10\n";

    return 0;
}
```

The above code executes identically to the following (which may be easier to understand):

```cpp
#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
    {
        if (x < 10)
            std::cout << x << "is less than 10\n";
        else
            std::cout << x << "is exactly 10\n";
    }

    return 0;
}
```

First, x > 10 is evaluated. If true, then the "is greater" output executes and we're done. Otherwise, the else statement executes. That else statement is a nested if statement. So we then check if x < 10. If true, then the "is less than" output executes and we're done. Otherwise, the nested else statement executes. In that case, we print the "is exactly" output, and we're done.

In practice, we typically don't nest the chained-ifs inside blocks, because it makes the statements harder to read (and the code quickly becomes nested if there are lots of chained if statements).

Here's another example:

```cpp
#include <iostream>

int main()
{
    std::cout << "Enter a positive number between 0 and 9999: ";
    int x;
    std::cin >> x;

    if (x < 0)
        std::cout << x << " is negative\n";
    else if (x < 10)
        std::cout << x << " has 1 digit\n";
    else if (x < 100)
        std::cout << x << " has 2 digits\n";
    else if (x < 1000)
        std::cout << x << " has 3 digits\n";
    else if (x < 10000)
        std::cout << x << " has 4 digits\n";
    else
        std::cout << x << " was larger than 9999\n";

    return 0;
}
```

**Nesting if statements**

It is also possible to nest if statements within other if statements:

```
#include <iostream>

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

    if (x >= 10) // outer if statement
        // it is bad coding style to nest if statements this way
        if (x <= 20) // inner if statement
            std::cout << x << "is between 10 and 20\n";

        // which if statement does this else belong to?
        else
            std::cout << x << "is greater than 20\n";

    return 0;
}
```

The above program introduces a source of potential ambiguity called a **dangling else** problem. Is the *else statement* in the above program matched up with the outer or inner *if statement*?

The answer is that an *else statement* is paired up with the last unmatched *if statement* in the same block. Thus, in the program above, the *else* is matched up with the inner *if statement*.

To avoid such ambiguities when nesting complex statements, it is generally a good idea to enclose the statement within a block. Here is the above program written without ambiguity:

```
#include <iostream>

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

    if (x >= 10)
    {
        if (x <= 20)
            std::cout << x << "is between 10 and 20\n";
        else // attached to inner if statement
            std::cout << x << "is greater than 20\n";
    }

    return 0;
}
```

Now it is much clearer that the *else statement* belongs to the inner *if statement*.

Encasing the inner *if statement* in a block also allows us to explicitly attach an *else* to the outer *if statement*:

```cpp
#include <iostream>

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

    if (x >= 10)
    {
        if (x <= 20)
            std::cout << x << "is between 10 and 20\n";
    }
    else // attached to outer if statement
        std::cout << x << "is less than 10\n";

    return 0;
}
```

The use of a block tells the compiler that the *else statement* should attach to the *if statement* before the block. Without the block, the *else statement* would attach to the nearest unmatched *if statement*, which would be the inner *if statement*.

**Using logical operators with if statements**

You can also have if statements check multiple conditions together by using the logical operators (covered in section **3.6 -- logical operators**):

```cpp
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y;
    std::cin >> y;

    if (x > 0 && y > 0) // && is logical and -- checks if both conditions are true
        std::cout << "both numbers are positive\n";
    else if (x > 0 || y > 0) // || is logical or -- checks if either condition is true
        std::cout << "One of the numbers is positive\n";
    else
        std::cout << "Neither number is positive\n";

    return 0;
}
```

**Common uses for if statements**

*If statements* are commonly used to do error checking. For example, to calculate a square root, the value passed to the square root function should be a non-negative number:

```cpp
#include <iostream>
#include <cmath> // for sqrt()

void printSqrt(double value)
{
    if (value >= 0.0)
        std::cout << "The square root of " << value << " is " << sqrt(value) << "\n";
    else
        std::cout << "Error: " << value << " is negative\n";
}
```

*If statements* can also be used to do **early returns**, where a function returns control to the caller before the end of the function. In the following program, if the parameter value is negative, the function returns a symbolic constant or enumerated value error code to the caller right away.

```
#include <iostream>

enum class ErrorCode
{
    ERROR_SUCCESS = 0,
    ERROR_NEGATIVE_NUMBER = -1
};

ErrorCode doSomething(int value)
{
    // if value is a negative number
    if (value < 0)
       // early return an error code
        return ErrorCode::ERROR_NEGATIVE_NUMBER;

    // Do whatever here

    return ErrorCode::ERROR_SUCCESS;
}

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

    if (doSomething(x) == ErrorCode::ERROR_NEGATIVE_NUMBER)
    {
        std::cout << "You entered a negative number!\n";
    }
    else
    {
        std::cout << "It worked!\n";
    }

    return 0;
}
```

*If statements* are also commonly used to do simple math functionality, such as a min() or max() function that returns the minimum or maximum of its parameters:

```
int min(int x, int y)
{
    if (x > y)
        return y;
    else
        return x;
}
```

Note that this last function is so simple, it can also be written using the conditional operator (?:):

```
int min(int x, int y)
{
    return (x > y) ? y : x;
}
```

**Null statements**

It is possible to omit the statement part of an *if statement*. A statement with no body is called a **null statement**, and it is declared by using a single semicolon in place of the statement. For readability purposes, the semicolon of a *null statement* is typically placed on its own line. This indicates that the use of a *null statement* was intentional, and makes it harder to overlook the use of the *null statement*.

```
if (x > 10)
```

```
    ; // this is a null statement
```

Null statements are typically used when the language requires a statement to exist but the programmer doesn't need one. We'll see examples of intentional null statements later in this chapter, when we cover loops.

*Null statements* are rarely used in conjunction with *if statements*. However, they often unintentionally cause problems for new or careless programmers. Consider the following snippet:

```
if (x == 0);
    x = 1;
```

In the above snippet, the user accidentally put a semicolon on the end of the *if statement*. This unassuming error actually causes the above snippet to execute like this:

```
if (x == 0)
    ; // the semicolon acts as a null statement
x = 1; // and this line always gets executed!
```

*Warning: Make sure you don't accidentally "terminate" your if statements with a semicolon.*

**Operator== vs Operator= inside the conditional**

Just a reminder that inside your if statement conditional, you should be using operator== when testing for equality, not operator= (which is assignment). Consider the following program:

```
#include <iostream>

int main()
{
    std::cout << "Enter 0 or 1: ";
    int x;
    std::cin >> x;
    if (x = 0) // oops, we used an assignment here instead of a test for equality
        std::cout << "You entered 0";
    else
        std::cout << "You entered 1";

    return 0;
}
```

This program will compile and run, but will always produce the result "You entered 1" even if you enter 0. This happens because x = 0 first assigns 0 to x, then evaluates to the value 0, which is boolean false. Since the conditional is always false, the else statement always executes.

**5.3 -- Switch statements**

**Index**

**5.1 -- Control flow introduction**