8.3 — Common if statement problems

ALEX SEPTEMBER 11, 2023

This lesson is a continuation of lesson 8.2 -- If statements and blocks (https://www.learncpp.com/cpp-tutorial/if-statements-and-blocks/). In this lesson, we'll take a look at some common problems that occur when using `if statements`.

Nested if statements and the dangling else problem

It is 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 >= 0) // 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 0 and 20\n";

    return 0;
}
```

Now consider the following program:

```
#include <iostream>

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

    if (x >= 0) // 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 0 and 20\n";

    // which if statement does this else belong to?
    else
        std::cout << x << " is negative\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`, as if the program had been written like this:

```
#include <iostream>

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

    if (x >= 0) // outer if statement
    {
        if (x <= 20) // inner if statement
            std::cout << x << " is between 0 and 20\n";
        else // attached to inner if statement
            std::cout << x << " is negative\n";
    }

    return 0;
}
```

This causes the above program to produce incorrect output:

```
Enter a number: 21
21 is negative
```

To avoid such ambiguities when nesting `if statements`, it is a good idea to explicitly enclose the inner `if statement` within a block. This allows us to attach an `else` to either `if statement` without ambiguity:

```
#include <iostream>

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

    if (x >= 0)
    {
        if (x <= 20)
            std::cout << x << " is between 0 and 20\n";
        else // attached to inner if statement
            std::cout << x << " is greater than 20\n";
    }
    else // attached to outer if statement
        std::cout << x << " is negative\n";

    return 0;
}
```

The `else statement` within the block attaches to the inner `if statement`, and the `else statement` outside of the block attaches to the outer `if statement`.

Flattening nested if statements

Nested `if statements` can often be flattened by either restructuring the logic or by using logical operators (covered in lesson 6.7 -- Logical operators (https://www.learncpp.com/cpp-tutorial/logical-operators/)). Code that is less nested is less error prone.

For example, the above example can be flattened as follows:

```
#include <iostream>

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

    if (x < 0)
        std::cout << x << " is negative\n";
    else if (x <= 20) // only executes if x >= 0
        std::cout << x << " is between 0 and 20\n";
    else // only executes if x > 20
        std::cout << x << " is greater than 20\n";

    return 0;
}
```

Here's another example that uses logical operators to check multiple conditions within a single `if statement`:

```
#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;
}
```

Null statements

A **null statement** is an expression statement that consists of just a semicolon:

```
if (x > 10)
    ; // this is a null statement
```

`Null statements` do nothing. They are typically used when the language requires a statement to exist but the programmer doesn't need one. For readability, `null statements` are typically placed on their own lines.

We'll see examples of intentional `null statements` later in this chapter, when we cover loops. `Null statements` are rarely intentionally used with `if statements`. However, they can unintentionally cause problems for new (or careless) programmers. Consider the following snippet:

```
if (nuclearCodesActivated());
    blowUpTheWorld();
```

In the above snippet, the programmer accidentally put a semicolon on the end of the `if statement` (a common mistake since semicolons end many statements). This unassuming error compiles fine, and causes the snippet to execute as if it had been written like this:

```
if (nuclearCodesActivated())
    ; // the semicolon acts as a null statement
blowUpTheWorld(); // and this line always gets executed!
```

Warning

Be careful not to "terminate" your `if statement` with a semicolon, otherwise your conditional statement(s) will execute unconditionally (even if they are inside a block).

Operator== vs Operator= inside the conditional

Inside your 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\n";
    else
        std::cout << "You entered 1\n";

    return 0;
}
```

This program will compile and run, but will produce the wrong result in some cases:

```
Enter 0 or 1: 0
You entered 1
```

In fact, this program will always produce the result `You entered 1`. This happens because `x = 0` first assigns the value `0` to `x`, then evaluates to the value of `x`, which is now `0`, which is Boolean `false`. Since the conditional is always `false`, the `else statement` always executes.

Next lesson
8.4 Constexpr if statements

Back to table of contents

Previous lesson
8.2 If statements and blocks