3.3 — Increment/decrement operators, and side effects

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 23RD, 2018

Incrementing (adding 1 to) and decrementing (subtracting 1 from) a variable are so common that they have their own operators in C. There are actually two versions of each operator -- a prefix version and a postfix version.

Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then evaluate x
Prefix decrement (pre-decrement)	—	—x	Decrement x, then evaluate x
Postfix increment (post-increment)	++	x++	Evaluate x, then increment x
Postfix decrement (post-decrement)	—	x—	Evaluate x, then decrement x

The prefix increment/decrement operators are very straightforward. The value of x is incremented or decremented, and then x is evaluated. For example:

```
int x = 5;
int y = ++x; // x is now equal to 6, and 6 is assigned to y
```

The postfix increment/decrement operators are a little more tricky. The compiler makes a temporary copy of x, increments or decrements the original x (not the copy), and then evaluates the temporary copy of x. The temporary copy of x is then discarded.

```
int x = 5;
int y = x++; // 5 is assigned to y, and x is now equal to 6
```

Let's examine how this last line works in more detail. First, the compiler makes a temporary copy of x that starts with the same value as x (5). Then it increments the original x from 5 to 6. Then the compiler evaluates the temporary copy, which evaluates to 5, and assigns that value to y. Then the temporary copy is discarded.

Consequently, y ends up with the value of 5, and x ends up with the value 6.

Here is another example showing the difference between the prefix and postfix versions:

```
int x = 5, y = 5;
std::cout << x << " " << y << '\n';
std::cout << ++x << " " << --y << '\n'; // prefix
std::cout << x << " " << y << '\n';
std::cout << x++ << " " << y-- << '\n'; // postfix
std::cout << x << " " << y << '\n';
```

This produces the output:

```
5 5
6 4
6 4
6 4
7 3
```

On the third line, x and y are incremented/decremented before they are evaluated, so their new values are printed by cout. On the fifth line, a temporary copy of the original values (x=6, y=4) is sent to cout, and then the original x and y are incremented. That is why the changes from the postfix operators don't show up until the next line.

Rule: Favor pre-increment and pre-decrement over post-increment and post-decrement. The prefix versions are not only more performant, you're less likely to run into strange issues with them.

Side effects

A function or expression is said to have a **side effect** if it modifies some state (e.g. any stored information in memory), does input or output, or calls other functions that have side effects.

Most of the time, side effects are useful:

```
x = 5;
++x;
std::cout << x;
```

The assignment operator in the above example has the side effect of changing the value of x permanently. Even after the statement has finished executing, x will have the value 5. Operator++ has the side effect of incrementing x. The outputting of x has the side effect of modifying the console.

However, side effects can also lead to unexpected results:

```
int add(int x, int y)
{
    return x + y;
}

int main()
{
    int x = 5;
    int value = add(x, ++x); // is this 5 + 6, or 6 + 6?  It depends on what order your compiler evalua
tes the function arguments in

    std::cout << value; // value could be 11 or 12, depending on how the above line evaluates!
    return 0;
}
```

C++ does not define the order in which function arguments are evaluated. If the left argument is evaluated first, this becomes a call to add(5, 6), which equals 11. If the right argument is evaluated first, this becomes a call to add(6, 6), which equals 12! Note that this is only a problem because one of the argument to function add() has a side effect.

Here's another popular example:

```
int main()
{
    int x = 1;
    x = x++;
    std::cout << x;

    return 0;
}
```

What value does this program print? The answer is: it's undefined.

If the ++ is applied to x before the assignment, the answer will be 1 (postfix operator++ increments x from 1 to 2, but it evaluates to 1, so the expression becomes x = 1).

If the ++ is applied to x after the assignment, the answer will be 2 (this evaluates as x = x, then postfix operator++ is applied, incrementing x from 1 to 2).

There are other cases where C++ does not specify the order in which certain things are evaluated, so different compilers will make different assumptions. Even when C++ does make it clear how things should be evaluated, some compilers implement behaviors involving variables with side-effects incorrectly. These problems can generally *all* be avoided by ensuring that any variable that has a side-effect applied is used no more than once in a given statement.

Rule: Don't use a variable that has a side effect applied to it more than once in a given statement. If you do, the result may be undefined.

Please don't ask why your programs that violate the above rule produce results that don't seem to make sense. That's what happens when you write programs that have "undefined behavior". ☺

For more information on undefined behaviors, revisit the "Undefined Behavior" section of lesson **1.3 -- A first look at variables, initialization, and assignment**.