5.4 — Increment/decrement operators, and side effects

ALEX AUGUST 6, 2022

Incrementing and decrementing variables

Incrementing (adding 1 to) and decrementing (subtracting 1 from) a variable are both so common that they have their own operators.

Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then return x
Prefix decrement (pre-decrement)	--	--x	Decrement x, then return x
Postfix increment (post-increment)	++	x++	Copy x, then increment x, then return the copy
Postfix decrement (post-decrement)	--	x--	Copy x, then decrement x, then return the copy

Note that there are two versions of each operator -- a prefix version (where the operator comes before the operand) and a postfix version (where the operator comes after the operand).

The prefix increment/decrement operators are very straightforward. First, the operand is incremented or decremented, and then expression evaluates to the value of the operand. For example:

```
#include <iostream>

int main()
{
    int x { 5 };
    int y = ++x; // x is incremented to 6, x is evaluated to the value 6, and 6 is assigned to
y

    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

This prints:

```
6 6
```

The postfix increment/decrement operators are trickier. First, a copy of the operand is made. Then the operand (not the copy) is incremented or decremented. Finally, the copy (not the original) is evaluated. For example:

```
#include <iostream>

int main()
{
    int x { 5 };
    int y = x++; // x is incremented to 6, copy of original x is evaluated to the value 5, and 5 is assigned to
y

    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

This prints:

```
6 5
```

Let's examine how this line 6 works in more detail. First, a temporary copy of x is made that starts with the same value as x (5). Then the actual x is incremented from 5 to 6. Then the copy of x (which still has value 5) is returned and assigned to y. Then the temporary copy is discarded.

Consequently, y ends up with the value of 5 (the pre-incremented value), and x ends up with the value 6 (the post-incremented value).

Note that the postfix version takes a lot more steps, and thus may not be as performant as the prefix version.

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

```
#include <iostream>

int main()
{
    int x{ 5 };
    int 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';

    return 0;
}
```

This produces the output:

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

On the 8th line, we do a prefix increment and decrement. On this line, x and y are incremented/decremented before their values are sent to std::cout, so we see their updated values reflected by std::cout.

On the 10th line, we do a postfix increment and decrement. On this line, the copy of x and y (with the pre-incremented and pre-decremented values) are what is sent to std::cout, so we don't see the increment and decrement reflected here. Those changes don't show up until the next line, when x and y are evaluated again.

Best practice

Strongly favor the prefix version of the increment and decrement operators, as they are generally more performant, and 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 does anything that persists beyond the life of the function or expression itself.

Common examples of side effects include changing the value of objects, doing input or output, or updating a graphical user interface (e.g. enabling or disabling a button).

Most of the time, side effects are useful:

```
x = 5; // the assignment operator modifies the state of x
++x; // operator++ modifies the state of x
std::cout << x; // operator<< modifies the state of the
console
```

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 still have the value 5. Similarly with operator++, the value of x is altered even after the statement has finished evaluating. The outputting of x also has the side effect of modifying the state of the console, as you can now see the value of x printed to the console.

However, side effects can also lead to unexpected results:

```
#include <iostream>

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 evaluates the function arguments in

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

The C++ standard 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 arguments to function add() has a side effect.

As an aside...

The C++ standard intentionally does not define these things so that compilers can do whatever is most natural (and thus most performant) for a given architecture.

There are other cases where the C++ standard does not specify the order in which certain things are evaluated (such as operator operands), so different compilers may exhibit different behaviors. Even when the C++ standard does make it clear how things should be evaluated, historically this has been an area where there have been many compiler bugs. 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.

Warning

C++ does not define the order of evaluation for function arguments or operator operands.

Warning

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.

Next lesson
5.5 Comma and conditional operators

Back to table of contents

Previous lesson
5.3 Modulus and Exponentiation