5.4 — Increment/decrement operators, and side effects

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY ALEX ON SEPTEMBER 19TH, 2019

Incrementing and decrementing variables

Incrementing (adding 1 to) and decrementing (subtracting 1 from) a variable are 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

    std::cout << x << ' ' << y;
    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

    std::cout << x << ' ' << y;
    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:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x{ 5 };
6      int y{ 5 };
7      std::cout << x << " " << y << '\n';
8      std::cout << ++x << " " << --y << '\n'; // prefix
9      std::cout << x << " " << y << '\n';
10     std::cout << x++ << " " << y-- << '\n'; // postfix
11     std::cout << x << " " << y << '\n';
12
13     return 0;
14 }
```

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:

```
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; // 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 arguments to function add() has a side effect.

There are other cases where C++ does not specify the order in which certain things are evaluated (such as operator operands), so different compilers may exhibit different behaviors. Even when C++ 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.

Best practice

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.

5.5 -- Comma and conditional operators

Index

5.3 -- Modulus and Exponentiation

C++ TUTORIAL | PRINT THIS POST

190 comments to 5.4 — Increment/decrement operators, and side effects

« Older Comments 1 2 3

HolzstockG
January 6, 2020 at 4:06 am · Reply

So as I understand "side effect" is nothing but change in states of already defined variables and functions?

nascardriver
January 7, 2020 at 4:26 am · Reply

Right

HolzstockG
January 9, 2020 at 12:28 am · Reply

I had hard time to understand side effect since it seemed to be too obvious xD (and it was).

Bowie
December 22, 2019 at 10:19 pm · Reply

if prefix is recommended, then why there are so many people still using postfix in for loop.
usually i see something like this:

```
for (int i=0; i<5; i++) {
   std::cout << i;
}
```

Alex
January 1, 2020 at 7:58 pm · Reply

1) Habit.
2) Because some people don't realize they actually do different things.
3) Because C++ makes it seem like postfix is natural. It should have been named ++C.

Renan
January 7, 2020 at 8:15 am · Reply

I really enjoyed the 3rd reason. Thanks for all the great work.

jfzcoman
December 18, 2019 at 6:20 pm · Reply

Hi,Alex,I've got confused after saw following form http://a4academics.com/interview-questions/57-c-plus-plus/725-c-interview-questions-for-freshers?showall=&start=2
23) What will be output of the following code snippet?

```
int num1=5;
int num2=5;
num1 =num1++ + num1--;
num2 =++num2 + --num2;
printf("num1=%d num2=%d",num1,num2);
```

Output will be num1=10 num2=10.

is this right? I thought it wouble num1=11

nascardriver
December 19, 2019 at 2:41 am · Reply

No, it's wrong. The code produces undefined behavior, the question doesn't have a correct answer.

5irKawfycups
November 5, 2019 at 6:35 pm · Reply

I feel like there may be a typo, or maybe I am misunderstanding? In the section "Side effects", the explanation under the first code block

```

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

Here, part of the explanation reads: "Even after the statement has finished executing, x will still have the value 5."

Wouldn't a reference to x now return 6, not 5? For that matter, wouldn't x++ also cause a later reference to x to return 6?

nascardriver
November 6, 2019 at 4:21 am · Reply

Hi!

The sentence you quoted refers to `x = 5`, not the other 2 statements.

```
x = 5;
// x is 5 here, even though the above statement finished

++x; // x is changed to 6
// and it's still 6 here
```

That's what the sentence is trying to say.

Avijit Pandey
September 24, 2019 at 12:38 pm · Reply

I have a very weird question that was handed to me by my professor, it's actually in C but I'm posting the C++ equivalent here since the outputs and the problems I'm encountering remain the same.

```
//Q: What should the following program print and how?
#include <iostream>

int main()
{
    int a = 1, b = 1, d = 1;
    std::cout << ++a + ++a + a++ << " " << a++ + ++b << " " << ++d + d++ + a++ << "\n";
    return 0;
}
```

I'm assuming here that when we do

```
(++a) + (++a) + (a++)
```

it evaluates as 2 + 3 + x. For x, if we assigned a new variable-the value of 'a++', it would be assigned 3 and 'a' would be incremented. But since we are directly adding a++, it's adding the value at the memory address of 'a' after evaluation of the expression 'a++'which will be 4. So the output should be 9. And using similar logic, the final Output should be 9, 6, 10.
However, the actual output is 15, 4 and 6.

Now, obviously, my evaluation order is not being followed by the compiler. But I can not figure out how on earth the first expression evaluates to 15!

2.One other weird thing that I noticed is that when I define 3 pointer variables to hold the memory addresses of 'a', 'b', and 'd' respectively(and never modify them), the output magically changes to 14, 4, and 5.

I do have a feeling I'm making it way too complicated than it needs to be, but I am seriously annoyed by this one line of code. Can you guys explain this please?

nascardriver
September 25, 2019 at 2:45 am · Reply

The output is unspecified in C++ and C, assuming that the original code is

```c
#include <stdio.h>

int main()
{
    int a = 1, b = 1, d = 1;
    printf("%i %i %i\n", ++a + ++a + a++, a++ + ++b, ++d + d++ + a++);
    //                   (1)              (2)        (3)
    return 0;
}
```

For one, the order of evaluation of the arguments is undefined, ie. it could be
(1) -> (2) -> (3)
(2) -> (1) -> (3)
(3) -> (1) -> (2)
...

I'm fairly sure that each part (1), (2), and (3) produces undefined behavior too before C++11 and in C. I'm not sure about C++11 onward, because sequencing rules changed.

The question cannot be answered.

> But I can not figure out how on earth the first expression evaluates to 15!
Undefined behavior. You code might as well change your wallpaper.

2.
Undefined behavior.

Your professor can't expect an answer other than that the code produces undefined behavior. This code never produced well defined output.

Avijit Pandey
September 25, 2019 at 3:12 am · Reply

I was discarding this answer because I thought it'd be way too obvious. Turns out it's true, thanks!

Nikola
September 13, 2019 at 9:24 pm · Reply

Hello,

Not technically important but there is a mistake in the first sentence of the "Side effects" section.

A function or expression is said to have a side effect if IT does

Alex
September 19, 2019 at 10:26 am · Reply

Fixed. Thanks!

Attila
August 13, 2019 at 6:07 am · Reply

I feel like the third paragraph from the bottom should be outlined as a "Rule" shown in lecture 0.2

Alex

August 13, 2019 at 11:55 am · Reply

This will be fixed soonish when I rollout the stylistic updates for this chapter.

Murat

August 6, 2019 at 4:03 am · Reply

```
1    int x = 1;
2        // x = x++;
3    // x = ++x;
4    std::cout << x;
```

When running both commented lines (individually), the output is 2. But you said that it would be 1 or 2 (depending on which is commented out).

nascardriver

August 6, 2019 at 6:25 am · Reply

Pre C++17, both lines cause undefined behavior. There's no point in talking about what the result could be, because anything could happen.

As I understand it, since C++17 this is guaranteed

```
1  int i{ 1 };
2
3  i = i++;
4
5  std::cout << i << '\n'; // 1
6
7  i = ++i;
8
9  std::cout << i << '\n'; // 2
```

Either I am wrong or gcc and clang are generating invalid warnings. Maybe someone else can clear up. My assumption is based on rule 20 here https://en.cppreference.com/w/cpp/language/eval_order#Rules

Hadal

August 28, 2019 at 1:30 pm · Reply

Are you saying that // are the results?
Because I put your code in VS and the results are 2 and 3.

nascardriver

August 28, 2019 at 11:15 pm · Reply

You need to enable a higher standard version in your project's settings.
msvc v19.22 produces 1 and 2 with /std:c++17. As I said in my comment, before c++17 behavior is undefined.

DecSco

July 22, 2019 at 4:24 am · Reply

Does this produce undefined behaviour?

```
#include <iostream>

int getIdentifier()
{
    static int identifier {0};
    return identifier++;
}

int main()
{
    std::cout << "1st call: ID = " << getIdentifier() << "\n";
    std::cout << "2nd call: ID = " << getIdentifier() << "\n";
    std::cout << "3rd call: ID = " << getIdentifier() << "\n";
    std::cout << "4th call: ID = " << getIdentifier() << "\n";
    return 0;
}
```

On my machine, it works fine. But I'm not sure that's universal, given that the execution may already have returned to the function call.

nascardriver
July 22, 2019 at 5:43 am · Reply

This is well-defined.

> the execution may already have returned to the function call
Mind elaborating? Your program is executed sequentially, it can't run 2 things at the same time.

DecSco
July 23, 2019 at 3:03 am · Reply

Cheers. That's what I needed to know.
What I meant is that at the time when the return statement gets executed, and the value of "identifier" is supposed to be returned to the caller, it still has to be incremented. So if the execution were to return to the caller at the point at which the value is correct, then the increment would be lost, because the execution has already left the function.
So the compiler would actually have to implement it as something like

```
return ++identifier -1;
```

Correct?

nascardriver
July 23, 2019 at 3:25 am · Reply

Side effect are applied when the function ends, `main` won't run before `identifier` has been incremented. You'll only lose information if you modify a variable multiple times in 1 statement, but you're not doing that.

WSLaFleur
May 24, 2019 at 5:53 pm · Reply

I found these examples really amusing and informative. Thanks for the lesson.

daileyj93
April 8, 2019 at 8:48 am · Reply

"For more information on undefined behaviors, revisit the “Undefined Behavior” section of lesson 1.3 -- Introduction to variables."

It looks like undefined behavior is now lesson 1.6 -- Uninitialized variables and undefined behavior, not lesson 1.3 anymore.

Alex
April 8, 2019 at 3:36 pm · Reply

Fixed. Thanks for pointing this out!

zakqux
February 8, 2019 at 5:07 pm · Reply

```
1 If the ++ is applied to x before the assignment, the answer will be 1 (postfix operator++ in
```

Can you clarify this? Don't understand quite how x=1...

nascardriver
February 9, 2019 at 7:02 am · Reply

postfix++ returns the value the variable had before it has been increased.
++prefix returns the value the variable has after having been increased.

```
1 int i{ 0 };
2 int iZak{ ++i };
3 // iZak = 1
4 // i = 1
5 int iQux{ i++ };
6 // iQux = 1
7 // i = 2
```

Both variants increase the variable by one, only the return value is different. Use ++prefix unless you need postfix++. postfix++ is slower.

Jörg
February 5, 2019 at 10:44 pm · Reply

Hi, I heard that c++17 introduced some changes to the evaluation order when working with side effects. Do you know any of this? Is

```
1 int i{5};
2 i = ++i + ++i; or
3 i = ++i + i++;
```

still undefined?

Alex
February 7, 2019 at 7:24 pm · Reply

It did introduce some evaluation order changes, and some of the expressions that yielded undefined results now resolve determinately. I don't know if those particular expressions are affected or not. The best practice doesn't change though -- don't use a variable with a side effect applied to it more than once in an expression.

Senna
February 3, 2019 at 2:20 am · Reply

Hi,
I've read previously lessons, even though they're updated do I gotta study them again ?

Alex
February 4, 2019 at 8:16 pm · Reply

For the lesson in the new chapters 1-3, no, but there's a lot of good stuff added, so you'd probably benefit from it if you did.

The previous lessons 2 and 3 (now D.2 and O.3) haven't been updated yet. They're next.

Senna
February 5, 2019 at 1:41 am · Reply

So they've updated to C++17 , right ?
Thanks anyway.

Alex
February 7, 2019 at 6:06 am · Reply

Where applicable.

« Older Comments 1 2 3