

## 5.2 — Arithmetic operators

ALEX AUGUST 2, 2021

### Unary arithmetic operators

There are two unary arithmetic operators, plus (+), and minus (-). As a reminder, unary operators are operators that only take one operand.

Operator	Symbol	Form	Operation
Unary plus	+	+x	Value of x
Unary minus	-	-x	Negation of x

The unary minus operator returns the operand multiplied by -1. In other words, if  $x = 5$ ,  $-x$  is -5.

The unary plus operator returns the value of the operand. In other words, +5 is 5, and  $+x$  is  $x$ . Generally you won't need to use this operator since it's redundant. It was added largely to provide symmetry with the *unary minus* operator.

For best effect, both of these operators should be placed immediately preceding the operand (e.g. `-x`, not `- x`).

Do not confuse the *unary minus* operator with the *binary subtraction* operator, which uses the same symbol. For example, in the expression `x = 5 - -3;`, the first minus is the *binary subtraction* operator, and the second is the *unary minus* operator.

### Binary arithmetic operators

There are 5 binary arithmetic operators. Binary operators are operators that take a left and right operand.

Operator	Symbol	Form	Operation
Addition	+	$x + y$	x plus y
Subtraction	-	$x - y$	x minus y
Multiplication	*	$x * y$	x multiplied by y
Division	/	$x / y$	x divided by y
Modulus (Remainder)	%	$x \% y$	The remainder of x divided by y

The addition, subtraction, and multiplication operators work just like they do in real life, with no caveats.

Division and modulus (remainder) need some additional explanation. We'll talk about division below, and modulus in the next lesson.

### Integer and floating point division

It is easiest to think of the division operator as having two different “modes”.

If either (or both) of the operands are floating point values, the *division operator* performs floating point division. Floating point division returns a floating point value, and the fraction is kept. For example, `7.0 / 4 = 1.75`, `7 / 4.0 = 1.75`, and `7.0 / 4.0 = 1.75`. As with all floating point arithmetic operations, rounding errors may occur.

If both of the operands are integers, the *division operator* performs integer division instead. Integer division drops any fractions and returns an integer value. For example, `7 / 4 = 1` because the fractional portion of the result is dropped. Similarly, `-7 / 4 = -1` because the fraction is dropped.

## Using `static_cast` to do floating point division with integers

The above raises the question -- if we have two integers, and want to divide them without losing the fraction, how would we do so?

In lesson 4.11 -- [Chars](#), we showed how we could use the `static_cast<>` operator to convert a char into an integer so it would print as an integer rather than a character.

We can similarly use `static_cast<>` to convert an integer to a floating point number so that we can do *floating point division* instead of *integer division*. Consider the following code:

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

This produces the result:

```
int / int = 1
double / int = 1.75
int / double = 1.75
double / double = 1.75
```

The above illustrates that if either operand is a floating point number, the result will be floating point division, not integer division.

## Dividing by zero

Trying to divide by 0 (or 0.0) will generally cause your program to crash, as the results are mathematically undefined!

```

1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Enter a divisor: ";
6      int x{};
7      std::cin >> x;
8
9      std::cout << "12 / " << x << " = " << 12 / x <<
10     '\n';
11
12     return 0;
13 }

```

If you run the above program and enter 0, your program will either crash or terminate abnormally. Go ahead and try it, it won't harm your computer.

## Arithmetic assignment operators

Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

Up to this point, when you've needed to add 4 to a variable, you've likely done the following:

```
1 | x = x + 4; // add 4 to existing value  
  | of x
```

This works, but it's a little clunky, and takes two operators to execute (operator+, and operator=).

Because writing statements such as `x = x + 4` is so common, C++ provides five arithmetic assignment operators for convenience. Instead of writing `x = x + 4`, you can write `x += 4`. Instead of `x = x * y`, you can write `x *= y`.

Thus, the above becomes:

```
1 | x += 4; // add 4 to existing value  
  | of x
```



**Next lesson**

**5.3** Modulus and Exponentiation



**Back to table of contents**



**Previous lesson**

**5.1** Operator precedence and associativity

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code`

 Name\*

 Email\* 

Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies:



POST COMMENT

DP N N FOUT

Newest ▼

