

6.2 — Arithmetic operators

 learncpp.com/cpp-tutorial/arithmetic-operators/

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 readability, 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
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 remainder need some additional explanation. We'll talk about division below, and remainder 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 [4.12 -- Introduction to type conversion and static_cast](#), 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:

```
#include <iostream>

int main()
{
    constexpr int x{ 7 };
    constexpr int y{ 4 };

    std::cout << "int / int = " << x / y << '\n';
    std::cout << "double / int = " << static_cast<double>(x) / y << '\n';
    std::cout << "int / double = " << x / static_cast<double>(y) << '\n';
    std::cout << "double / double = " << static_cast<double>(x) / static_cast<double>
(y) << '\n';

    return 0;
}
```

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 0 and 0.0

Integer division with a divisor of 0 will cause undefined behavior, as the results are mathematically undefined!

```
#include <iostream>

int main()
{
    constexpr int apples { 12 };
    std::cout << "You have " << apples << " apples. Enter how many people to
divide them between: ";
    int x {};
    std::cin >> x;

    std::cout << "Each person gets " << apples / x << " whole apples.\n"; //
apples and x are int, so this is integer division

    return 0;
}
```

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

The result of dividing by floating point value 0.0 is implementation-defined (meaning the behavior is determined by the compiler/architecture). On architectures that support IEEE754 floating point format, the result will be NaN or Inf. On other architectures, the result will likely be undefined behavior.

Related content

We discuss NaN and Inf in lesson [4.8 -- Floating point numbers](#).

You can see what your program does by running the following program and entering 0 or 0.0:

```
#include <iostream>

int main()
{
    constexpr int apples { 12 };
    std::cout << "You have " << apples << " apples. Enter how many servings of
apples you want: ";

    double d {};
    std::cin >> d;

    std::cout << "Each serving is " << apples / d << " apples.\n"; // d is
double, so this is floating point division

    return 0;
}
```

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
Remainder 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:

```
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:

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