3.2 — Arithmetic operators

BY ALEX ON JUNE 13TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 31ST, 2017

Unary arithmetic operators

There are two unary arithmetic operators, plus (+), and minus (-). If you remember, 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 plus operator returns the value of the operand. In other words, +5 = 5, and +x = 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.

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

For best effect, both of these operators should be placed immediately preceding the operand (eg. `-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 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.

Integer and floating point division

It is easiest to think of the division operator as having two different "modes". If both of the operands are integers, the division operator performs integer division. Integer division drops any fractions and returns an integer value. For example, 7 / 4 = 1 because the fraction is dropped. Note that integer division does not round.

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 / 3 = 2.333, 7 / 3.0 = 2.333, and 7.0 / 3.0 = 2.333.

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

Using static_cast<> to do floating point division with integers

In section **2.7 -- Chars**, we showed how we could use the static_cast<> operator to cast a char to an integer so it would print correctly.

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
```

```
int main()
{
    int x = 7;
    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
```

Modulus (remainder)

The modulus operator is also informally known as the remainder operator. The modulus operator only works on integer operands, and it returns the remainder after doing integer division. For example, 7 / 4 = 1 remainder 3, thus 7 % 4 = 3. As another example, 25 / 7 = 3 remainder 4, thus 25 % 7 = 4.

Modulus is very useful for testing whether a number is evenly divisible by another number: if x % y == 0, then we know that y divides evenly into x.

For example, say we wanted to write a program that printed every number from 1 to 100 with 20 numbers per line. We could use the modulus operator to determine where to do the line breaks. Even though you haven't seen a while statement yet, the following program should be pretty comprehensible:

```
#include <iostream>

int main()
{
    // count holds the current number to print
    int count = 1; // start at 1

    // Loop continually until we pass number 100
    while (count <= 100)
    {
        std::cout << count << " "; // print the current number

        // if count is evenly divisible by 20, print a new line
        if (count % 20 == 0)
            std::cout << "\n";

        count = count + 1; // go to next number
    } // end of while

    return 0;
} // end of main()
```

This results in:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

We'll discuss while statements in a future lesson.

A warning about integer division and modulus with negative numbers prior to C++11

Prior to C++11, if either of the operands of integer division are negative, the compiler is free to round up or down! For example, -5 / 2 can evaluate to either -3 or -2, depending on which way the compiler rounds. However, most modern compilers truncate towards 0 (so -5 / 2 would equal -2). The C++11 specification changed this to explicitly define that integer division should always truncate towards 0 (or put more simply, the fractional component is dropped).

Also prior to C++11, if either operand of the modulus operator is negative, the results of the modulus can be either negative or positive! For example, -5 % 2 can evaluate to either 1 or -1. The C++11 specification tightens this up so that a % b always resolves to the sign of a.

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 5 to a variable, you've likely done the following:

```
x = x + 5;
```

This works, but it's a little clunky, and takes two operators to execute.

Because writing statements such as x = x + 5 is so common, C++ provides 5 arithmetic assignment operators for convenience. Instead of writing x = x + 5, you can write x += 5. Instead of x = x * y, you can write x *= y.

Where's the exponent operator?

Astute readers will note that there is no exponent operator. Instead, you have to use the pow() function that lives in the cmath library. pow(base, exponent) is the equivalent of $base^{exponent}$. It's worth noting that the parameters to pow() are doubles, so it can handle non-integer exponents.

```
#include <cmath> // needed for pow()
#include <iostream>

int main()
{
    std::cout << "Enter the base: ";
    double base;
    std::cin >> base;

    std::cout << "Enter the exponent: ";
    double exp;
    std::cin >> exp;

    std::cout << base << "^" << exp << " = " << pow(base, exp) << "\n";

    return 0;
}
```

Quiz

1) What does the following expression evaluate to? 6 + 5 * 4 % 3

2) Write a program that asks the user to input an integer, and tells the user whether the number is even or odd. Write a function called isEven() that returns true if an integer passed to it is even. Use the modulus operator to test whether the integer parameter is even.

Hint: You'll need to use if statements and the comparison operator (==) for this program. See lesson **2.6 -- Boolean values and an introduction to if statements** if you need a refresher on how to do this.

Answers

1) **Hide Solution**

Because * and % have higher precedence than +, the + will evaluate last. We can rewrite our expression as 6 + (5 * 4 % 3). Operators * and % have the same precedence, so we have to look at the associativity to resolve them. The associativity for operators * and % is left to right, so we resolve the left operator first. We can rewrite our expression like this: 6 + ((5 * 4) % 3).

6 + ((5 * 4) % 3) = 6 + (20 % 3) = 6 + 2 = 8

2) **Hide Solution**

```
#include <iostream>

bool isEven(int x)
{
    // if x % 2 == 0, 2 divides evenly into our number, which means it must be an even number
    return (x % 2) == 0;
}

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    if (isEven(x))
        std::cout << x << " is even\n";
    else
        std::cout << x << " is odd\n";

    return 0;
}
```

Note: You may have been tempted to write function isEven() like this:

```
bool isEven(int x)
{
    if ((x % 2) == 0)
        return true;
    else
        return false;
}
```

While this works, it's more complicated than it needs to be. Let's take a look at how we can simplify it. First, let's pull out the if statement conditional and assign it to a separate boolean:

```
bool isEven(int x)
{
    bool isEven = (x % 2) == 0;
    if (isEven) // isEven is true
        return true;
    else // isEven is false
        return false;
}
```

Now, note that the if statement above essentially says "if isEven is true, return true, otherwise (if isEven is false) return false". If that's the case, we can just return isEven:

```
bool isEven(int x)
{
    bool isEven = (x % 2) == 0;
    return isEven;
}
```

And in this case, since we only use variable isEven once, we might as well eliminate the variable:

```
bool isEven(int x)
{
    return (x % 2) == 0;
}
```



3.3 -- Increment/decrement operators, and side effects


Index


3.1 -- Operator precedence and associativity

C++ TUTORIAL | PRINT THIS POST

182 comments to 3.2 — Arithmetic operators

« Older Comments 1 2 3

Aditi
July 14, 2018 at 8:45 pm · Reply

My solution

```
#include "stdafx.h"
#include <iostream>
using namespace std;
bool isEven(int a) {
    if (a % 2 == 0) {
        return true;
    }
    else {
        return false;
    }


}
int main()
{
    int a;
    cout << "Enter an Integer" << endl;
    cin >> a;
    if (isEven(a)) {
        cout << "Integer is Even" << endl;
    }
    else
        cout << "Integer is Odd" << endl;
    return 0;
}
```