02. Integer Values

Where's your pencil?

Objectives

- Use integer literals
- Use integer operators
- Use associative and precedence rules to evaluate an integer expression correctly
- Use operators for integer values according to associative and precedence rules
- Write arithmetic expressions according to standard C++ practices

In this set of notes, we will learn to print integer values and work with some integer operators.

Beyond hello world

So far we have been printing textual data (i.e. C-strings) onto the console window. It's time to do some math. We'll start off with whole numbers.

For this set of notes, instead of writing

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;

    return 0;
}
```

I will write:

```
std::cout << "Hello, world!" << std::endl;
```

You should be smart enough to fill in the “surrounding” code:

```
#include <iostream>

int main()
{


    return 0;
}
```

By the way, it's a good practice to end your program output with a newline.

Printing integers

An **integer** is just a whole number.

Try this:

```
#include <iostream>

int main()
{
    std::cout << 42 << std::endl;
    std::cout << -5000 << std::endl;

    return 0;
}
```

Now this:

```
#include <iostream>

int main()
{
    std::cout << 42 << -5000 << std::endl;

    return 0;
}
```

and then this:

```
#include <iostream>

int main()
{
    std::cout << 42 << ", " << -5000 << std::endl;

    return 0;
}
```

AHA! So you can print integers and strings in the same print statement.

Exercise. Write a C++ program that produces this output:

```
Life, universe and everything ... 42!
```

You must use the ***integer*** 42 in your program.

Exercise. Write a C++ program that produces this output:

```
6 x 7 = 42
```

You must use the ***integers*** 6, 7, and 42 in your program.

Here's an important jargon. Look at our program again:

```
#include <iostream>

int main()
{
    std::cout << 42 << ", " << -5000 << std::endl;

    return 0;
}
```

The integer value 42 in the program is called an **integer literal**. Sometimes we also call this an **integer constant**. The phrase “in the program” is important. So if you're in your math classes, and you write

42

in your assignment, you don't call that an integer literal!!!

Integer operators: +, -, and *

Now you might say, “Isn't this ...”

```
std::cout << 42;
```

the same as

```
std::cout << "42";
```

Yes. The ***output*** is the same. But of course the **value** printed out in the first program is an integer while the second is a string.

This will REALLY show you the difference. Try this:

```
std::cout << 42 + 1 << std::endl;
std::cout << "42 + 1" << std::endl;
```

Get it?

Soooo ... C++ can do math!!!

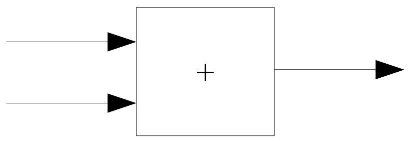
Try this:

```
std::cout << "42 + 1 = " << 42 + 1 << std::endl;
std::cout << "42 - 1 = " << 42 - 1 << std::endl;
std::cout << "2 * 3 = " << 2 * 3 << std::endl;
```

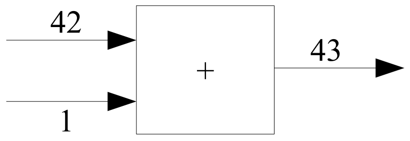
So far we see that C++ understands three **operators**: +, `-`, and `*`.

No surprises there ... I hope!!!

You can (and should) think of the integer operator + as some kind of machine that accepts two integer inputs and returns to you an integer value:



And here's what happens when you put 42 and 1 into this machine:



In C++, this operator + machine accepts integer inputs. There's another operator + machine that accepts for instance numbers with decimal places. The two are ***totally different*** machines. This is extremely important. (I'll talk about numbers with decimal places in another set of notes. These are called floating point numbers.) If I want to emphasize, I will call the + in this set of notes, I will call it the **integer + operator**.

The story is very similar for operator – and operator *.

Exercise. Write a C++ program that produces this output

```
6 x 7 = 42
```

In your program, the only integers you can use are 6 and 7 and the strings you can use are `" x "` and `" = "`. (Do not use characters at all; for instance do not use `'4'`).

Exercise. Can C++ understand and operate with negative numbers? Write a C++ program that prints the following (skip a line after printing each integer):

- The addition of -1 and -3.
- The subtraction 5 from 2.
- The subtraction of -3 from -5.
- The product of -3 and -2.
- The product of 3 and -2.

Check that the output gives the expected values.

Integer operators: / and %

Division is ... well try this

```
std::cout << "4 / 2 = " << 4 / 2 << std::endl;
```

And then this:

```
std::cout << "100 / 20 = " << 100 / 20 << std::endl;
```

So far so good. No surprises.

BUT ... what about this program:

```
std::cout << "9 / 3 = " << 9 / 3 << '\n'
          << "8 / 3 = " << 8 / 3 << '\n'
          << "7 / 3 = " << 7 / 3 << std::endl;
```

So you see what's happening?

If a and b are integer, then a / b gives the **quotient** of a and b. This is also called the **integer division** of a by b. This is the same as losing the fractional part of the usual mathematical division.

Let me repeat that again ...

In C++, `13 / 3` gives the **integer part** of 13 divided by 3. In math, 13 divided by 3 is 4.333333... In C++, when `/` is used as an operator for two **integer** values in a C++ program, the result is an **integer**. So you lose the fractional part of the result from mathematical division.

Exercise. In C++ notation (do NOT use your C++ compiler) what is `20 / 10`? What about `21 / 10`? Now verify using your C++ compiler with this:

```
std::cout << 20 / 10 << '\n'
          << 21 / 10 << std::endl;
```

REMEMBER THAT!

When / is applied to two integers, we say that this division is an **integer division**. (Of course since we call this an *integer* division operator you would expect the division operator in 1.123 / 3.2343 to be something totally different. We'll talk about this later.)

An integer division will produce an integer except for this case:

```
std::cout << 1 / 0 << std::endl;
```

Run it. You will get an error. I hope that's not surprising! Division by zero will cause an error. You already know that from your math classes: you cannot divide by zero.

Exercise. First work out on a piece of paper (or in your head) the values of the following

- `10 + 55`
- `25 - 50`
- `11 * 2`
- `25 / 4`
- `(-25) / 4`
- `25 / (-4)`
- `(-25) / (-4)`

Now verify your computation using C++.

Something like 10 + 55 is called an **expression** (or integer expression since it contains only integers). Here's another integer expression:

$$10 + 55 + 412 - 12 * 43 - 67$$

Computing the resulting value of the expression is called **evaluating** the expression.

But what about the fractional part that's lost when doing an integer division? What if you really wanted it?

Consider 13 / 3. C++ will give you 4. You're losing one-third. (Right?) In math, the correct answer is 4 and one-third:

$$\frac{13}{3} = 4\frac{1}{3}$$

The 1 above is called the **remainder**. Here's another example:

$$\frac{43}{8} = 5\frac{3}{8}$$

i.e., when 43 is divided by 8, you get 5 with a remainder of 3. What this means is very simple. If you have 43 dollar bills and you want to give them equally to your 8 good friends, then each will get 5 dollars and you're left with 3 dollars. Let me repeat that – when you're given

$$\frac{43}{8}$$

then in order to write

$$\frac{43}{8} = 5\frac{3}{8}$$

the 5 is the integer division of 43 by 8 and the 3 is the remainder when 43 is divided by 8.

Exercise. Using pen-and-paper technology compute the remainder of the following:

- 10 is divided by 3
- 9 is divided by 3
- 2 is divided by 3

Verify your computation by running this program:

```
std::cout << 10 % 3 << std::endl;
std::cout << 9 % 3 << std::endl;
std::cout << 2 % 3 << std::endl;
```

First of all you see that `%` is an operator. What does the operator `%` do?

Answer: ____

`%` is called the **mod** operator.

Exercise. Using pen-and-paper technology

- Compute the quotient and remainder when 10 is divided by 3
- Compute the quotient and remainder when 25 is divided by 4
- Compute the quotient and remainder when 72 is divided by 23

Now verify using C++.

The integer quotient and remainder operators occur frequently in real life. They are not just some fancy academic fluff. For instance suppose you are told: “It took John 135 minutes to paint this wall,” but you prefer to think in terms of hours and minutes. What would you do? You would do this (mathematically):

$$\frac{135}{60} = 2\frac{15}{60}$$

i.e.,

135 minutes = 2 hours and 15 minutes

To ***check*** that this is correct:

2 hours + 15 minutes
= 2 x 60 minutes + 15 minutes
= 120 minutes + 15 minutes
= 135 minutes

Correct? Of course the 2 is the integer quotient 135 / 60 (in C++) and the

15 is integer mod 135 % 60 (in C++). And why do we quotient and mod ***by 60***? Because 1 hour equals 60 minutes. Correct?

Exercise. Check that 192 minutes is 3 hours and 12 minutes, i.e., check if

$$\frac{192}{60} = 3\frac{12}{60}$$

But that's not the whole story. The remainder operator (i.e., the mod or the % operator) is in fact crucial to cryptography. Without it, e-commerce would have been impossible. That's how important it really is. I bet your high school teacher didn't tell you that.

Exercise. Convert 1435 minutes to hours and minutes. Do not use a calculator – use C++.

Exercise. Convert 1435 hours to days and hours. (1 day = 24 hours.) Do not use a calculator – use C++.

Exercise. You're 75 (OK … you're not … pretend you're 75) and you have decided to divide your wealth $13,524 equally to 7 grandchildren. You're not a cheapskate so you're only going to give out whole dollars and not cents. You're going to keep the remainder to yourself. How much will you be left with after dividing your wealth? Verify with C++.

Exercise. Convert 143546 seconds to days, hours, minutes, and seconds.

Exercise. The time is now 3:35PM. After working on his C++ notes for 134 minutes, how many minutes does John have to wait till dinnertime at 6PM?

Exercise. On Mars, 11 makes up a dezon. For instance a dezon of eggs is 11 eggs. 15 eggs would be 1 dezon and 4 eggs. 100 eggs would be

100 ________ and 100 _________ eggs

(in each of the blanks write down one operator and one value; the second resulting value must be less than 11.)

Exercise. On Mars, 8 dezons make a nozed. For instance 15 eggs would be 1 dezon and 4 eggs. 90 eggs would be 1 nozed, 0 dezon, and 2 eggs. 100 eggs would be

100 _______ nozed, 100 _______ dezon, and 100 _______ eggs

(in each blank write one operator and one value; the second resulting value must be less than 8 and the third resulting value must be less than 11.)

Remember this:

```
std::cout << 5 % 0 << std::endl;
```

Question: What are you to remember? And this too:

```
std::cout << 5 / 0 << std::endl;
```

Using integer mod % (and the integer quotient above) to convert from minutes to hours and minutes might give you the impression that they are use only for very trivial things. WRONG!!! Without the concept of integer mod operator, we would not have the explosion of internet e-commerce. Modern encryption (and many other things!!!) depends on the concept of the integer mod operation.

Altogether we see that C++ understands five integer operators:

+	addition
–	subtraction
*	multiplication
/	division (or quotient)
%	remainder

These operators are **binary** in the sense that you need to feed two integers to the operator in order to get a result.

FOCUS! Read the above paragraphs again.

Exercise. QUICKLY! **Close** your notes and answer the following question: What is a binary operator? Write down 5 binary integer operators in C++.

You should thank the designers of C++ that they use the operator signs +, – , *, / which you are used to, except for a slight change in meaning to /. Whenever possible, programming language designers try to make programming easier by using common mathematical notation and convention.

The dangerous ^ operator

I want to give you **a very important warning**.

You know from using your TI graphing calculator that ^ is used for exponentiation. For instance if you want two-to-the-power-of-four, 2^4, mathematically this is 2x2x2x2, you enter

```
2^4
```

into your calculator and it will spit out 16 for you. (Right?) Now run this program:

```
std::cout << (2 ^ 4) << std::endl;
```

When you run it ... it gives ... *drumroll* ...

```
6
```

and not 16!!! Whoa!!! Wassup???

The reason is not that C++ can't do exponentiation. In C++ the ^ has a different meaning. We will talk about that much later. The important thing to remember right now is that **^ is not exponentiation in C++**.

Exercise. Write a C++ program that prints the following:

```
8
```

Your program cannot use any string or character, and the integer that can appear in your program is the integer 2 and operators. (There are two ways to do this.)

Exercise. What is 2^0?

Exercise. Compute 2^i for i = 0, 1, 2, 3, 4, 5, 6, 7, 8,9,10 by hand. Memorize these powers!

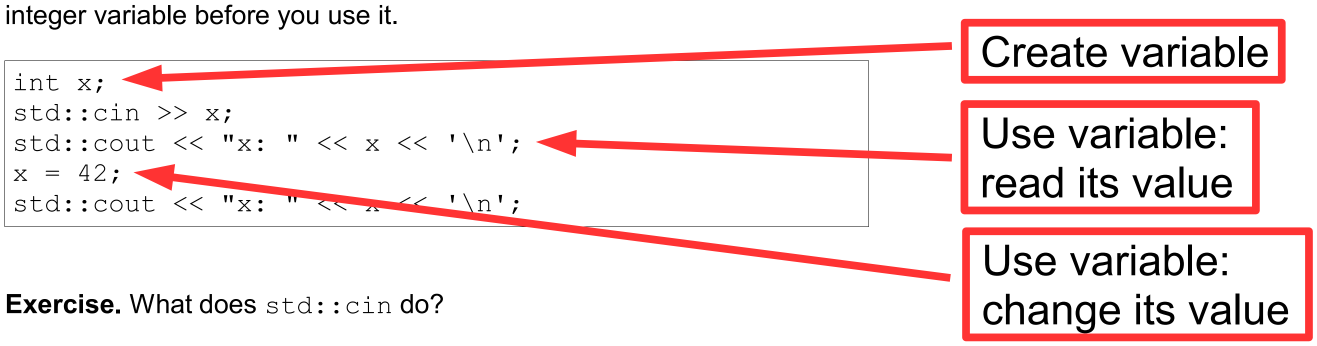
Variables: input and output

Try this:

```
int x;
std::cin >> x;
std::cout << "x: " << x << '\n';
x = 42;
std::cout << "x: " << x << '\n';
```

When you run the program, enter an integer value on your keyboard and press the Enter key.

`x` is an **integer variable**. `x` is a **variable** because … it's a variable – it holds a value and the value can vary. `x` is an **integer** variable because it can only hold an integer value. You have to create an integer variable before you use it.



```
int x;
std::cin >> x;
std::cout << "x: " << x << '\n';
x = 42;
std::cout << "x: " << x << '\n';
```

Exercise. What does `std::cin` do?

It's not too surprising that you can do this:

```
int x;
std::cin >> x;
std::cout << "x: " << x << '\n';
std::cout << "x + 1: " << x + 1 << '\n';
```

You can create more than one variable. Try this one:

```
int x, y;
std::cin >> x;
std::cout << "x: " << x << '\n';
std::cout << "x + 1: " << x + 1 << '\n';
int z;
std::cin >> y >> z;
std::cout << "y + z: " << y + z << '\n';
```

I'll come back to variables again in more details … there's a whole set of notes on variables.

Exercise. Write a program that gets two integers from the user and prints the product of the two inputs. You have 2 minutes.

Exercise. Write a program that gets three integers from the user and prints the sum of the three inputs. You have 2 minutes.

A few simple facts: divisors and primes

Divisors and Remainders

Divisors and primes and prime factorizations are already covered in elementary algebra. Here's a quick review of some facts on divisors tied in to C++.

An integer d is a **divisor** of another n if d divides n. In other words an integer d is a divisor of n if you can find an integer x such that mathematically:

dx = n

For instance 3 is a divisor of 60 since mathematically

3x = 60

for x = 20. On the other hand 3 is not a divisor of 61 since you cannot find an ***integer*** x satisfying

3x = 61

Another way to think of it is that

d is a divisor of n if the quotient of n by d gives a 0 remainder

i.e.,

d is a divisor of n if n % d is 0

For the case of 61, when you divide 61 by 3 you get

$$61 = 2\frac{1}{3}$$

i.e. the remainder is 1, i.e.,

61 % 3 is 1 which is not 0

So 3 is not a divisor of 61, or 61 is not divisible by 3.

Exercise. Using C++, print 61 % 3.

Exercise. Is 1 a divisor of 100? Is 1 a divisor of 1354597? Is 1 a divisor of -1234312? Is 1 a divisor of all integers?

Exercise. Is 34 a divisor of 1352? Is 34 a divisor 1768?

Of course you know that if you have something like this for positive integer n:

n = dx

where d and x are positive integers greater than 1, you can think of that as breaking up n into two pieces (or factors) d and x. You can continue to do that for both d and x until you get the prime factorization of n. For instance for n = 100,

100 = 4 x 25

This breaks up 100 into two factors 4 and 25. We now factorize 4 as

4 = 2x2

and so

100 = (2x2)x25

We are not done since we can factorize 25 as

25 = 5x5

This gives us the following factorization of 100:

100 = (2x2)x(5x5)

Since 2 and 5 are primes you can factorize them into smaller positive integers greater than 1. So we get the prime factorization of 100:

100 = 2x2x5x5.

And you should know that the prime factorization of any positive integer greater than 1 into primes is **unique**. For instance in the above example we started with 100 and factorize 100 into 4x25. We then factorize 4 and 25 separately until we get

100 = 2x2x5x5.

If we started the factorization with 50x2 and then factorize 50 and 2 separately we will still arrive

50 = 2x5x5

and therefore

100 = 50x2 = (2x5x5)x2 = 2x5x5x2.

After you've organized the above prime factorizations in so that the terms are ascending order we get

100 = 2x2x5x5
100 = 2x5x5x2 = 2x2x5x5

If I use exponentiation notation, I can write this:
100 = 2^2 x 5^2

Therefore no matter how you perform the prime factorization, after rearranging the prime factors in ascending order, the prime factorization is always the same. This fact is called the **fundamental theorem of arithmetic** which states that …

All positive integers can be expressed as a product of primes and, up to rearranging the primes, there is only one such prime factorization.

Exercise. Find the prime factorization of 935. First do this by hand. Second do it with the help of C++. Don't use your calculator.

Exercise. Find the prime factorization of 51315. First do this by hand. Second do it with the help of C++. Don't use your calculator.

Of course we just used the concept of primes. But what's a prime? Here's a quick review. A **prime** is a positive integer greater than 1 that is divisible only by 1 and itself. (Note that by definition 1 is not a prime.) For instance 13 is a prime. 42 on the other hand is not a prime since for instance 2 divides 42:

42 = 2 x 21

Of course 3 divides 42 as well since

42 = 3 x 14

But in order to say that 42 is not a prime, you only need to find ***<u>one</u>*** positive divisor of 42 that is not 1 and not 42.

To check that 7 is a prime, you need to check
- 2 is not a divisor of 7
- 3 is not a divisor of 7
- 4 is not a divisor of 7
- 5 is not a divisor of 7
- 6 is not a divisor of 7

which is the same as checking:

- The remainder when 7 divided by 2 is not 0
- The remainder when 7 divided by 3 is not 0
- The remainder when 7 divided by 4 is not 0
- The remainder when 7 divided by 5 is not 0
- The remainder when 7 divided by 6 is not 0

which is the same as checking

- 7 % 2 is not 0
- 7 % 3 is not 0
- 7 % 4 is not 0
- 7 % 5 is not 0
- 7 % 6 is not 0

Make sure you follow the above sequence of translations of prime checking down to a sequence of % operator checks!!!

Exercise. Check if 7 is a prime. (You need to complete the following.)

- 7 % 2 is ___ therefore 2 ______ (is or is not) a divisor of 7
- 7 % 3 is ___ therefore 3 ______ (is or is not) a divisor of 7
- 7 % 4 is ___ therefore 4 ______ (is or is not) a divisor of 7
- 7 % 5 is ___ therefore 5 ______ (is or is not) a divisor of 7
- 7 % 6 is ___ therefore 6 ______ (is or is not) a divisor of 7

(Use C++ for the first blank for each line) Therefore you conclude that 7 is a prime.

To check if 11 is a prime, you will need to

- 2 is not a divisor of 11
- 3 is not a divisor of 11
- 4 is not a divisor of 11
- 5 is not a divisor of 11
- 6 is not a divisor of 11
- 7 is not a divisor of 11
- 8 is not a divisor of 11
- 9 is not a divisor of 11
- 10 is not a divisor of 11

Again, you're trying to hunt down a potential divisor of 11 between 2 and 10 (1 less than 11).

Exercise. To check if 11 is a prime, you need to complete the following:

- 11 % 2 is ___ therefore 2 ______ (is or is not) a divisor of 11
- 11 % 3 is ___ therefore 3 ______ (is or is not) a divisor of 11
- 11 % 4 is ___ therefore 4 ______ (is or is not) a divisor of 11
- 11 % 5 is ___ therefore 5 ______ (is or is not) a divisor of 11
- 11 % 6 is ___ therefore 6 ______ (is or is not) a divisor of 11
- 11 % 7 is ___ therefore 7 ______ (is or is not) a divisor of 11
- 11 % 8 is ___ therefore 8 ______ (is or is not) a divisor of 11
- 11 % 9 is ___ therefore 9 ______ (is or is not) a divisor of 11
- 11 % 10 is ___ therefore 10 ______ (is or is not) a divisor of 11

(Use C++ for the first blank of each line) Therefore you conclude that 11 is a prime. [Yes, it's important … do this exercise.]

Exercise. To check if 17 is a prime using the above method.

Now suppose we attempt to check if 35 is a prime, we would do the same:

- 35 % 2 is ___ therefore 2 ______ (is or is not) a divisor of 35
- 35 % 3 is ___ therefore 3 ______ (is or is not) a divisor of 35
- 35 % 4 is ___ therefore 4 ______ (is or is not) a divisor of 35
- 35 % 5 is ___ therefore 5 ______ (is or is not) a divisor of 35
- 35 % 6 is ___ therefore 6 ______ (is or is not) a divisor of 35
- 35 % 7 is ___ therefore 7 ______ (is or is not) a divisor of 35
- etc.

WAIT!!!! HOLD YOUR HORSES!!!

When we were testing integer division by 5, you see that 35 % 5 is 0. This means that 5 ***is*** a divisor of 35. That means that 35 is (already) ***not*** a prime. There's no point in continuing the checks!!! I should have stopped then!!! Why waste time?!?

Exercise. To check if 21 is a prime (yes, yes, yes … it's not a prime … I know … nonetheless … follow the given procedure), you need to complete the following:

- 21 % 2 is ___ therefore 2 ______ (is or is not) a divisor of 21
- 21 % 3 is ___ therefore 3 ______ (is or is not) a divisor of 21
- 21 % 4 is ___ therefore 4 ______ (is or is not) a divisor of 21
- 21 % 5 is ___ therefore 5 ______ (is or is not) a divisor of 21
- 21 % 6 is ___ therefore 6 ______ (is or is not) a divisor of 21
- 21 % 7 is ___ therefore 7 ______ (is or is not) a divisor of 21
- 21 % 8 is ___ therefore 8 ______ (is or is not) a divisor of 21
- 21 % 9 is ___ therefore 9 ______ (is or is not) a divisor of 21
- 21 % 10 is ___ therefore 10 ______ (is or is not) a divisor of 11

(Use C++ for the first blank of each line). Stop the check AS SOON AS POSSIBLE. Is 21 a prime?

Exercise. Is 1351 a prime? Use the above method. Stop the check AS SOON AS POSSIBLE.

Coding style for expressions

For binary operators:

BAD:

```
123+13-32*234/23%228
```

BAD:

```
123+ 13 -32*   234/   23   %228
```

GOOD

```
123 + 13 - 32 * 234 / 23 % 228
```

Enough said.

For unary operators:

BAD

```
+  123 + + 13 - - 32 * 234
```

GOOD

```
+123 + +13 - -32 * 234
```

Enough said.

Incidentally, you **cannot** write this:

```
+123 ++ 13 -- 32 * 234
```

Because ++ and `--` have special meanings in C++. We'll talk about that later.

A few simple facts: 10

There's something special about integer division by 10 and mod 10.

Try this:

```
std::cout << "1358 / 10 = " << 1358 / 10 <<std::endl;
```

You notice that 1358 / 10 is just 1358 with its rightmost digit chopped off, i.e., it's 135.

That makes sense right? *Mathematically* 1358 divided by 10 gives 135.8. So in *C++*, since you only get the integer part of the division, the integer division 1358 / 10 is 135.

Exercise. Complete the following

```
std::cout << 1358 / __________ << std::endl;
```

so that the output is leftmost 2 digits of 1358, i.e., the output is

```
13
```

Exercise. Complete the following

```
std::cout << 4325456 / __________ << std::endl;
```

so that the output is leftmost 4 digits of `4325456`, i.e., the output is

```
4325
```

The rightmost digit of 1358 that was chopped off can be calculated using the mod (i.e. %) operator. Run this:

```
std::cout << "1358 % 10 = " << 1358 % 10 <<std::endl;
```

Exercise. Complete the following

```
std::cout << 1358 % __________ << std::endl;
```

so that the output is rightmost 4 digits of `1358`, i.e., the output is

```
58
```

Exercise. Complete the following

```
std::cout << 4325456 / __________ << std::endl;
```

so that the output is rightmost 2 digits of `4325456`, i.e., the output is

```
56
```

Exercise. Complete the following

```
std::cout << 7365459 / __________ << std::endl;
```

so that the output is rightmost 3 digits of `7365459`, i.e., the output is

```
459
```

Exercise. Complete the following using only 2 integer operators and 2 integers so that the output is the *second rightmost digit* of 1358 (i.e. 5):

```
std::cout << 1358 __ __ __ __ << std::endl;
```

The output is

```
5
```

Exercise. Complete the following using only 2 integer operators and 2 integers:

```
std::cout << 135246 __ __ __ __ << std::endl;
```

so that the output is

```
52
```

The above exercises are important: Extracting data from data is very common in Computer Science (well ... it's important in every area of Science). For instance google extracts keywords from web pages for indexing so that when someone does search for that word, google can return the web page's URL quickly. But that's slightly different from our case since google extracts data from strings and not integers.

The above involves ***cutting out*** digits of chunks of digits from an integer value. You can ***build*** integer values using chunks of integers. For instance you know from your math classes that

1358

is the same as (using mathematical notation):

1x1000 + 3x100 + 5x10 + 8x1

In this case, you are constructing 1358 using the digits 1, 3, 5 and 8 together with powers of 10. (Don't forget that 1 is a power of 10 too … 10^0 = 1. Correct?)

Exercise. Do this on your own:

```
std::cout << 1 * 1000 + 3 * 100 + 5 * 10 + 8 * 1
          << std::endl;
```

Do you get 1358?

Note that multiplying a number with a power of 10 is the same as adding a certain number of zeroes to the left of the given number. For instance in the above,

3x100 = 300

i.e., multiplication by 100 just adds 2 zeroes to the given number. Here's another example:

8 x 10000 = 80000

This works not just for digits (of course). For instance

867 x 100000 = 86700000

Exercise. Complete the following

```
std::cout << 5 * _____ + 8 * ______ + 2 * ______
          << std::endl;
```

so that the output is

```
582
```

Exercise. Complete the following

```
std::cout << 3 * _____ + 0 * ______ + 5 * _____
          << std::endl;
```

so that the output is

```
305
```

Exercise. Complete the following

```
std::cout << 123 * _____ + 456 * _____
          << std::endl;
```

so that the output is

```
123456
```

Exercise. Complete the following

```
std::cout << 123 * _____ + 456 * _____
          << std::endl;
```

so that the output is

```
12300456
```

Exercise. Complete the following

```
std::cout << 123 * _____ + 456 * _____
          << std::endl;
```

so that the output is

```
12756
```

A few simple facts: evenness/oddness

Another thing you should be aware of is the following:

First any integer % 5 is either 0 or 1 or 2 or 3 or 4. Right?

In general, if n is a positive integer, then any integer % n is either 0 or 1 or 2 or ... or n - 1.

In particular any integer % 2 is either 0 or 1. Note that an integer is even if its mod 2 is 0. Otherwise it's odd.

Try this:

```
std::cout << 5 % 2 << std::endl;
std::cout << 6 % 2 << std::endl;
```

Get it?

Associativity and precedence

College Algebra is a prerequisite. So you know what I mean by associativity and operator precedence right? Yes? No?

All the operators you saw (+, -, *, /, %) are binary operators in the sense that for instance + is applied to two values. For example we have 2 + 3.

Associativity.

Now think about this integer expression:

```
2 + 3 + 1
```

What we really mean is

```
(2 + 3) + 1
```

As you can see with the parentheses, each + is applied to two numbers.

- The left + is applied to 2 and 3.
- The right + is applied to (2+3) and 1, i.e., 5 and 1.

Correct?

But wait. There's another way to do it:

```
2 + (3 + 1)
```

So is 2 + 3 + 1

```
(2 + 3) + 1      or      2 + (3 + 1)      ???
```

Of course it doesn't matter!!! Ultimately they both evaluate to the same value:

```
(2 + 3) + 1              2 + (3 + 1)
= 5 + 1                  = 2 + 4
= 6                      = 6
```

But this is not the case for all operators. For instance, consider the minus operator:

```
(2 – 3) – 1      or      2 – (3 – 1)
= –1 – 1                 = 2 – 2
= –2                     = 0
```

Long time ago, mathematicians decided that they are too efficient to write

```
(2 + 3) + 1
```

So they got together and come to an agreement that if parentheses are left out for a string of pluses, you always do the pluses from left to right.

We say that **+ associates left-to-right** or **associates to the right**. So

1 + 2 + 3 + 4

is really a shorthand for

((1 + 2) + 3) + 4

The important thing to remember is that whether + is left or right associated is a matter of convention.

+, –, *, /, % associates from left to right. So for instance if I write

3 – 4 – 5 – 2

I mean

((3 – 4) – 5) – 2

i.e. do the leftmost subtraction first.

Precedence Rules

What about the case where you have not just pluses? What about something like

1 + 3 – 2 * 6 / 3 * 4 + 6

In C++, you determine which operator goes first using this table:

( )	first priority
*, /, %	second priority
+, –	third priority

Remember PEMDAS from your math classes? Note the difference is that * and / are at the same level, and + and – are at the same level.

Exercise. Stare hard at the above table for 5 seconds. Close your notes and reproduce it.

It will be clearer when I do an example. So let's evaluate the above expression:

1 + 3 – 2 * 6 / 3 * 4 + 6

When we check the priority table, we don't see parentheses, i.e., there's nothing for the first priority. What about the second priority operators? I do see * and /. So let's do them. Don't forget that these associate left-to-right which is just a fancy way of saying “do the leftmost first and keep moving right”. You see that for * and / (i.e., ignore + and – for now), the first to occur is the * here:

1 + 3 – $\underline{2 * 6}$ / 3 * 4 + 6

So let's do that:
1 + 3 – $\underline{2 * 6}$ / 3 * 4 + 6
= 1 + 3 – 12 / 3 * 4 + 6   i.e., 2 * 6 = 12

The next in the second priority operators is here:
1 + 3 – $\underline{12 / 3}$ * 4 + 6
Let's do that:
1 + 3 – $\underline{12 / 3}$ * 4 + 6
= 1 + 3 – 4 * 4 + 6   i.e., 12 * 3 = 4
Got it?

And the last evaluation for the second priority operators give us this:
1 + 3 – $\underline{4 * 4}$ + 6
= 1 + 3 – 16 + 6   i.e., 4 * 4 = 16

There are now no more second priority operators. Let's move on to the third priority operators, i.e., the + and the –. Here we go:

$\underline{1 + 3}$ – 16 + 6
= $\underline{4 – 16}$ + 6   i.e., 1 + 3 = 4
= $\underline{–12 + 6}$ i.e., 4 – 16 = -12
= –6   i.e., -12 + 6 = -12

Don't forget that + and – associates left-to-right. So we have to always pick + and – going left-to-right.

That's all there is to it. If I put all the computations together this is what I get:
1 + 3 – $\underline{2 * 6}$ / 3 * 4 + 6
= 1 + 3 – $\underline{12 / 3}$ * 4 + 6   i.e., 2 * 6 = 12
= 1 + 3 – $\underline{4 * 4}$ + 6   i.e., 12/ 3 = 4
= $\underline{1 + 3}$ – 16 + 6   i.e., 4 * 4 = 16
= $\underline{4 – 16}$ + 6   i.e., 1 + 3 = 4
= $\underline{–12 + 6}$ i.e., 4 – 16 = -12
= –6   i.e., -12 + 6 = -12

Exercise. Here's the same problem:
1 + 3 – 2 * 6 / 3 * 4 + 6
Redo the problem just like the way I did it. Explain every single step. BE EXACT, BE PRECISE, BE LOGICAL!!!

Associative and precedence are extremely important because performing computations in different orders can lead to different results. If you do it in the wrong order, then ... that's ***BAD!*** Just imagine the collapse of financial institutions or random explosions of missiles/shuttles because of a misunderstanding of the way C++ computes value!!!

Fortunately C++ is designed so that computations done in the computer follows the standard associative and precedence rules.

Exercise. Write a C++ program that prints 1 + 3 – 2 * 6 / 3 * 4 + 6. You have better get -6!!! Follow the procedure as above, explaining every single step. BE EXACT, BE PRECISE, BE LOGICAL!!!

Exercise. Evaluate this expression by hand (this means without the computer):

5 / 4 / 3

Verify your result with C++.

Exercise. Evaluate this C++ expression by hand

1 – 3 – 4 * 5 / 3 + 4 % 2

Verify your result with C++.

Exercise. Repeat the above (evaluate by hand and verify with C++) for these C++ expressions:

- 5 / 2 + 4 % 3 – 1 * 3
- 5 % 2 * 4 % 3 – 1 + 3
- 5 – 2 + 4 * 3 – 1 / 3

(If you can't get them all right in 1 minute, then you should create examples like these:

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

filling in random numbers and operators at the appropriate places and do the same (evaluate and verify) until you can get all of them right.

Let's try another example, this time an expression with a pair of parentheses:

3 + 1 * 4 – 4 * 3 * 2 / (2 + 6 * 2 – 1) + 6 * 7

In this case, there's one pair of parentheses. So we have to evaluate the expression within the parentheses first. In the parentheses, there are no parentheses, i.e., there are no priority one operators. So we look for priority two operators: * and /. There is one. So we evaluate that operator:

$3 + 1 * 4 – 4 * 3 * 2 / (2 + \underline{6 * 2} – 1) + 6 * 7$
$= 3 + 1 * 4 – 4 * 3 * 2 / (2 + 12 – 1) + 6 * 7$ i.e., 6 * 2 = 12

To make it easier to follow, I've underlined the expression I'm evaluating.

There are no priority two operators. So we look for priority 3 operators, i.e., + and –. There are two so we do this going left-to-right:

$3 + 1 * 4 – 4 * 3 * 2 / (\underline{2 + 12} – 1) + 6 * 7$
$= 3 + 1 * 4 – 4 * 3 * 2 / (\underline{14 – 1}) + 6 * 7$ i.e., 2 + 12 = 14

= 3 + 1 * 4 – 4 * 3 * 2 / (13) + 6 * 7	i.e., 14 – 1 = 13
= 3 + 1 * 4 – 4 * 3 * 2 / 13 + 6 * 7	i.e., (13) = 13

(In the last step we just remove the useless parentheses.) Now there are no parentheses (priority one). So look for priority two operators and evaluate them left-to-right:

$3 + \underline{1 * 4} - 4 * 3 * 2 / 13 + 6 * 7$	
$= 3 + 4 - \underline{4 * 3} * 2 / 13 + 6 * 7$	i.e., 1 * 4 = 4
$= 3 + 4 - \underline{12 * 2} / 13 + 6 * 7$	i.e., 4 * 3 = 12
$= 3 + 4 - \underline{24 / 13} + 6 * 7$	i.e., 12 * 2 = 24
$= 3 + 4 - 1 + \underline{6 * 7}$	i.e., 24 / 13 = 1
= 3 + 4 – 1 + 42	i.e., 6 * 7 = 42

Don't forget that in the fourth line, 24/13 = 1 because we're doing C++ integer quotient, not real number division in college algebra.
Now there are no more priority two operators. We evaluate the priority three operators, i.e., the + and –.

$\underline{3 + 4} - 1 + 42$	
$= \underline{7 - 1} + 42$	i.e., 3 + 4 = 7
$= \underline{6 + 42}$	i.e., 7 – 1 = 6
= 48	i.e., 6 + 42 = 48

and we're done.

Study every single step of the above computation.

Exercise. Here's the same problem again:
3 + 1 * 4 – 4 * 3 * 2 / (2 + 6 * 2 – 1) + 6 * 7
Redo the above computation without looking at the above solution. Make sure that every single step of your computations matches mine.

Exercise. Evaluate by hand showing all steps and verify with C++ the following C++ expressions:

- 10 / (2 + 4 % 2) – 1 * 3
- 10 % 2 * 4 % (2 – 1 + 3)
- 10 – (2 + 4 * 2) – 1 / 3

Exercise. Evaluate the integer expression

3 + 1 * 4 – 4 * 3 * 2 / (2 + 6 * 2 – 1) + 6 * 7

yourself using only the priority table and paper-and-pencil technology.

Exercise. Use C++ to evaluate 3 + 1 * 4 – 4 * 3 * 2 / (2 + 6 * 2 – 1) + 6 * 7 and make sure you get 48.

If you have an expression with ***two*** separate pairs of parentheses such as:

2 * (3 + 5) – 3 * (5 + 3 * 2)

you should present your answer by evaluating left-to-right, i.e., evaluate the leftmost parentheses first.

$2 * (\underline{3 + 5}) - 3 * (5 + 3 * 2)$	
$= 2 * \underline{(8)} - 3 * (5 + 3 * 2)$	i.e., 3 + 5 = 8
$= 2 * 8 - 3 * (5 + \underline{3 * 2})$	i.e., (8) = 8
$= 2 * 8 - 3 * (\underline{5 + 6})$	i.e., 3 * 2 = 6
$= 2 * 8 - 3 * \underline{(11)}$	i.e., 5 + 6 = 11
$= \underline{2 * 8} - 3 * 11$	i.e., (11) = 11
$= 16 - \underline{3 * 11}$	i.e., 2 * 8 = 16
$= \underline{16 - 33}$	i.e., 3 * 11
$= -17$	i.e., 16 – 33 = -17

Exercise. Evaluate
–1 – 2 * (3 + 4) / (2 * 4 + 5) + 6 % 3 / 2
by hand showing all steps. Next, verify your answer again a C++ program.

Exercise. Evaluate
3 + 2 * (3 % 5) + (1 + 4 / 5) + 2 * 4 – 2
by hand showing all steps. Next, verify your answer again a C++ program.

You can also have situations when parentheses are nested like this:

$1 + \underline{(2 * 1 + (3 + 4 * 5) / 2)} + 5$

i.e., within a pair of parentheses, you see another pair. As you're evaluating the first pair of parentheses, you have to apply the priority rules, which means you have to evaluate the inner parentheses first. That's all.

$1 + (2 * 1 + (3 + \underline{4 * 5}) / 2) + 5$	
$= 1 + (2 * 1 + (\underline{3 + 20}) / 2) + 5$	i.e., 4 * 5 = 20
$= 1 + (2 * 1 + \underline{(23)} / 2) + 5$	i.e., 3 + 20 = 23
$= 1 + (\underline{2 * 1} + 23 / 2) + 5$	i.e., (23) = 23
$= 1 + (2 + \underline{23 / 2}) + 5$	i.e., 2 * 1 = 2
$= 1 + (\underline{2 + 11}) + 5$	i.e., 23/ 2 = 11
$= 1 + \underline{(13)} + 5$	i.e., 2 + 11 = 13
$= \underline{1 + 13} + 5$	i.e., (13) = 13
$= \underline{14 + 5}$	i.e., 1 + 13 = 14
$= 19$	i.e., 14 + 5 = 19

Exercise. Evaluate 2 + 3 * 8 + 4 / 3 + 2 * (6 + (3 – 4) * 2) * 4 – 6 % 4 by hand showing all steps. Verify your answer with C++.

Exercise. Evaluate 2 + 3 * 8 + (4 / 3 + 2) * (6 + (3 – 4) * 2) * 4 – 6 % 4 by hand showing all steps. Verify your answer with C++.

Exercise. The expression has redundant parentheses:

((1 + 2) + (3 – 4)) * (3 % 2) + ((4) * 5) / 3

Without evaluating the operators, rewrite the expression into one requiring the least number of parentheses. Verify your work using C++. In other words print the above expression and then your simplified expression. If they are the same, you're good. If not, do the problem again.

Exercise. Do the same for this:

(1 + (2 * 3) – 4) * 3 % (2 * (9 % 3) / 3)

Exercise. What's wrong with this?

(1 + (2 * 3) – 4) * 3 % (2 * (4 % 5) / 3)

Exercise. What's wrong with this?

(5 + (3 + 4 – 4) * 3 – (2 * (4 % 1) / 3)

Unary and binary operators

An operator is **binary** if it is applied to two values. For instance the + in the following expression:

2 + 3

is binary. You have already seen the binary +, –,*,/,%. They all require two values to make sense.

There are actually two operators that require only one value. These are **unary** operators. The unary + operator (the "positive" operator) appears in this expression:

+2

+5 is 5

and of course there's the unary – operator (the "negative" operator):

–2

-(-3) is 3

So the symbol + and – have two different meanings.

Now we need to modify the chart of precedence rules:

( )	first priority
+, – (unary)	second priority
*, /, %	third priority
+, – (binary)	fourth priority

Exercise. Stare hard at the table for 5 seconds. Remember it. Close your notes and reproduce it. Continue until you get it correct three times in a row.

Try this example involving three unary – operators (the "negative" operator):

```
std::cout << - - - 2;
```

You should get -2. Right? If you try to put parentheses in the expression, of course the only reasonable way to do it would be like this:

```
-(-(-2))
```

I mean ... what else can it be??? You can't do this: `(((-)-)2)`!!! It has no meaning at all!!!

In other words you apply the rightmost `-` first. The last `-` is the leftmost. The unary `-` operator associates right to left. It's the same for the unary + operator ("positive" operator).

Summary

You can think of an **operator** as something that will give you a value.

A **binary** operator is an operator that produces a value when you feed it two values.

A **unary** operator is an operator that produces a value when you feed it one value.

An operator is an **integer** operator if it produces an integer value.

C++ understands the following binary integer operator:

+	addition
–	subtraction
*	multiplication
/	division
%	remainder

and the following unary operators:

+	positive
–	negative

Here are the precedence rules for the operators:

( )	first priority
+, – (unary)	second priority
*, /, %	third priority
+, – (binary)	fourth priority

The binary integer operators associates left-to-right. The unary integer operators associates right-to-left. So for example:

1 + 2 – 3 + 4 is ((1 + 2) – 3) + 4

and

-+-1 is -(+(-1))

You can print integer values in a print statement:

```
std::cout << 42 << std::endl;
```

You can mix the printing of integer, string values, and character values:

```
std::cout << 1 << ", " << 2 << "buckle my shoe"
          << '!' << std::endl;
```

When you print an expression, the evaluated expression is printed:

```
std::cout << 40 + 1 + 1 << std::endl;
```

Exercises.

1. What is the output of the following program? Do not use C++ yet.

```
std::cout << "(12 + 3) / 3" << std::endl;
```

Now, verify using C++. (If you said 5, go ahead and slap yourself and read this set of notes 100 times right away.)

2. What is the value of this C++ expression?

```
3 - 4 + 5 * 3 - (4 + 5) / 3
```

Verify using C++. (If you didn't get that right, write down 200 random expressions in C++ and spend the rest of the week evaluating all them.)

3. The following expression has too many parentheses. Rewrite it with the least number of parenthesis without evaluating the operators so that the resulting value is the same.

```
1 + (2 * 3) * 5 - (4 + (2 + (7 * 5))) % (2 + (3 - 1))
```

4. John and Doe made $135 selling their comics. John took out his laptop and ran this program:

```
#include <iostream>

int main()
{
    std::cout << 135 / 2 << std::endl;

    return 0;
}
```

and gave Doe the amount shown on the console window. Doe called John a crook. Why? Does Doe know C++?

5. Aunt Jane is giving away $1335235 to her 14256 nieces and nephews. She hates small change so she's going to give them whole dollar amounts. Furthermore to be impartial, all nieces and nephews will get the same amount. She will keep what remains. Since she has just taken CISS240 from Dr. Liow, she quickly took out her laptop and wrote a program that figured out how each should get (including herself). Duplicate her program. How much will each niece or nephew get? How much does Aunt Jane get to keep?

6. True of false? The annual compounding formula for principal P at a rate r after t years is given by:

$P(1 + r)^t$

Therefore the following C++ code will print the amount you have after putting $1000 into the national TrustMePlease bank that gives 300% annual interest (i.e. 3):

```
std::cout << 1000 * (1 + 3) ^ 5 << std::endl;
```

7. Check that the number 13 is prime using the procedure given in this set of notes.

8. Check that 30 is not a prime using the given procedure.