C++ Programming

Dr. Yihsiang Liow (May 24, 2025)

Contents

Chapter 0

Print statements and C-strings

OBJECTIVES

- Print strings and characters using `std::cout`
- Use the `\n`, `\t`, `\"`, `\'`, `\\` characters
- Use `std::endl` to force newline
- Write multiple print statements

Why the big, fat right margin? For your notes. Get busy.

In this set of notes, we learn to print strings and characters.

A quick advice to ease the pain of learning your first programming language: Type the program **exactly** as given. Even your spaces and blank lines must match the spaces and blank lines in my programs.

Let's begin ...

0.1 Hello World

Go ahead and run your first program:

```
#include <iostream>

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

    return 0;
}
```

... commercial break ... go to notes on software tool(s) for writing and running a program ...

Practice doing a hello world program until you can do it in < 2 minutes and without your notes.

Now let's go back to the C++ code. Right now, you should think of the stuff in bold as the program:

```
#include <iostream>

int main()
{
    *** YOUR "PROGRAM" GOES HERE ***

    return 0;
}
```

Therefore you can treat this as a template:

Enter a single program here.

```
#include <iostream>

int main()
{

    *** PROGRAM ***

    return 0;
}
```

```
#include <iostream>

int main()
{

    *** PROGRAM ***

    return 0;
}
```

TODO ... Put your program here

I will not explain things like "`#include <iostream>`" or "`int main()`" until later. (Technically, they are also part of the program.)

Try this

```
#include <iostream>

int main()
{
    std::cout << "Hello ... world! ... mom!\n";

    return 0;
}
```

Exercise 0.1.1. (Go to Solution 0.1.2 on page 6)
Write a program that gets the computer to greet you:

```
hello dr. liow, my name is c++.
```

(replace my name with yours ... you're probably not "`dr. liow`"). 2.5 minutes! ☐

Exercise 0.1.2. (Go to Solution 0.1.3 on page 6)
Debug (i.e., correct) this program by hand and then verify by running it with your C++ compiler. 3 minutes!

```
#include (iostream)

int main()
(
    std;;cout < "Hello, world!\n";

    return 0;
)
```

☐

SOLUTIONS

Solution 0.1.1. (Go to Exercise **??** on page **??**)

```
#include <iostream>

int main()
{
    std::cout << "hello columbia\n";

    return 0;
}
```

Solution 0.1.2. (Go to Exercise 0.1.1 on page 5)

```
#include <iostream>

int main()
{
    std::cout << "hello dr. liow, my name is c++.\n";

    return 0;
}
```

Solution 0.1.3. (Go to Exercise 0.1.2 on page 5)

```
#include ¡iostream¿

int main()
–
    std::cout <¡ "Hello, world!\n";

    return 0;
˝
```

0.2 Statement

Here's the first jargon. Look at our program again:

```
#include <iostream>

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

    return 0;
}
```

The line in bold is a **statement**. In C++, statements must terminate with a **semi-colon**. Note that the semi-colon is part of the statement.

At this point you should think of a statement as something that will cause your computer to perform some operation(s) when you run the program.

If you like, you can think of a statement as a sentence and the semi-colon as a period. When you're told to write a C++ statement, don't forget the semicolon!!! That would be like writing a sentence without a period (or question mark or exclamation mark) in an english essay.

Exercise 0.2.1. (Go to Solution 0.2.1 on page 9)
Modify your program by removing the first semicolon to get this:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n"

    return 0;
}
```

Run it. Does it work? Fix it. Test to make sure it works (that means: run the program.) □

Exercise 0.2.2. (Go to Solution 0.2.2 on page 9)
Replace the semicolons by periods.

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n".

    return 0.
}
```

Run it. Does it work? Fix it. ☐

See what I mean by this advice:

> *A quick advice to ease the pain of learning your first programming language: Type the program **exactly** as given. Even the spaces and blank lines must match my programs.*

Computers are dumb (and picky about details.) We are the smart ones. So . . . when you communicate with your computer, you have to be exact and explicit in your programs.

Exercise 0.2.3. (Go to Solution 0.2.3 on page 9)
Now write a program that prints any message (example: "do you want green eggs and ham?") ... close your notes first. Peek at your notes only when you have problems. ☐

SOLUTION

Solution 0.2.1. (Go to Exercise 0.2.1 on page 7)
No it won't work. You must have the semicolon. ☐

Solution 0.2.2. (Go to Exercise 0.2.2 on page 7)
Nope. It won't work either. You *must* have the semicolon. ☐

Solution 0.2.3. (Go to Exercise 0.2.3 on page 8)

```
#include <iostream>

int main()
{
    std::cout << "do you want green eggs and ham?\n";

    return 0;
}
```

☐

0.3 C-Strings

The stuff in quotes is called a C-string or just a string:

```
#include <iostream>

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

    return 0;
}
```

You can think of a string as textual data. (Soon I'll talk about numeric data for numeric computations. Got to have that for computer games, right?)

Double quotes are used to mark the beginning and ending of the string. So technically they are not part of the string. That's why when you run the above program, you do not see double-quotes.

Exercise 0.3.1. (Go to Solution 0.3.1) Does it work without the double-quotes?

```
#include <iostream>

int main()
{
    std::cout << Hello, world!\n;

    return 0;
}
```

□

In the string `"Hello, world!\n"`, `H` is a character. The next character is `e`. Etc. When we talk about characters we enclose them with single-quotes. So I should say character `'H'` rather than `H` or `"H"`. You can think of the character as the smallest unit of data in a string.

The string `"Hello, world!\n"` contains characters `'H'` and `'e'` and `'l'` and `'l'` and ... However a character such as `'H'` can contain one and only one unit of textual data. So you cannot say that `'Hello'` is a character.

Note that a string can contain as many as characters as you like: 10, 20, 50, 100, etc. There's actually a limit, but we won't be playing around with a string with 1,000,000 characters anyway for now – that would be a pain to type!!! Note that a string can contain no characters at all: `""`. Of course a string can contain exactly one character. For instance, here's a string with one character: `"H"`.

Note that `"H"` is a string with one character whereas `'H'` is a character. You just have to look at what quotes are used to tell if the thingy is a string or a character. That's all.

Exercise 0.3.2. (Go to Solution 0.3.2) Does it work with single-quotes?

```
#include <iostream>

int main()
{
    std::cout << 'Hello, world!\n';

    return 0;
}
```

□

Exercise 0.3.3. (Go to Soluton 0.3.3) Can you also print characters?? (You did think about this, right?) Try this:

```
#include <iostream>

int main()
{
    std::cout << '?';

    return 0;
}
```

□

Exercise 0.3.4. (Go to Solution 0.3.4) Which of the following is a string, a character, or neither:

```
"I"
like
{eggs}
"and"
'ham'
```

`'.'`

□

SOLUTION

Solution 0.3.1. (Go to Exercise 0.3.1) No! □

Solution 0.3.2. (Go to Exercise 0.3.2) No! □

Solution 0.3.3. (Go to Exercise 0.3.3) Yes. □

Solution 0.3.4. (Go to Exercise 0.3.4) Which of the following is a string, a character, or neither:

```
"I"      string
like     neither
'green'  neither
{eggs}   neither
"and"    string
'ham'    neither
'.'      character
```

□

0.4 Case Sensitivity

Is C++ case sensitive?

Exercise 0.4.1. Modify your program by changing `std` to `Std`:

```
#include <iostream>

int main()
{
    Std::cout << "Hello, world!\n";

    return 0;
}
```

Run it. Does it work? Fix it. ☐

Exercise 0.4.2. Modify your program by changing `return` to `RETURN`:

```
#include <iostream>

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

    RETURN 0;
}
```

Does it work? Fix it. ☐

Now answer this question:

Is C++ case sensitive? Circle one: YES NO

(duh ... I'm not taking answers in class.)

0.5 Whitespaces

A whitespace is ... well ... a white space.

Spaces, tabs, and newlines are whitespaces.

Exercise 0.5.1. Modify your program by inserting some spaces:

```
#include <iostream>

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

    return 0;
}
```

Does it work? ☐

Exercise 0.5.2. Modify your program by inserting some newlines:

```
#include <iostream>

int main()
{
    std::cout <<




            "Hello, world!\n";

    return 0;
}
```

☐

In general you can insert whitespaces between "basic words" understood by C++. These "basic words" are called **tokens**. (Oooooo another big word.)

If you think of statements as sentences, semi-colons as periods, then you can think of tokens as words.

For instance the following are some tokens from the above program: `int`, `return` and even {.

We say that C++ **ignores whitespace**.

Exercise 0.5.3. Try this:

```
#include <iostream>

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

    return 0;
}
```

Does it work? Now try this:

```
#include <iostream>

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

    return 0;
}
```

Does it work? □

The above shows you that **::** is a token – you cannot break it down.

Exercise 0.5.4. Try this:

```
#include <iostream>

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

    return 0;
}
```

Does it work? □

Try this:

```
#include <iostream>

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

    return 0;
}
```

The whitespaces between tokens are removed when your computer runs your C++ program. So, to the computer, it doesn't matter how much whitespace you insert between tokens – it still works.

However spaces in a ***string*** are not removed before the program runs. The space characters `' '` (there are 10 in the above string) are actually characters within the string.

Note that although you can insert whitespaces, in general, good spacing of a program makes it easier to read. Therefore you must follow the style of spacing shown in my notes. In particular, I **don't** want to see monstrous programs like this (although the program does work):

```
#include <iostream>

int                         main(){std
::
                     cout<<

"Hello, world!\n"; return 0; }
```

or this:

```
#include <iostream>
int main(){std::cout<<"Hello, world!\n";return 0;}
```

And don't try to be cute this like ...

```
#include <iostream>
int
 main
  ()
   {
    std
     ::
      cout
       <<
        "Hello, world!\n";
         return 0;
          }
```

The above examples are excellent candidates for the F grade. The correct coding style produces this:

```
#include <iostream>

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

    return 0;
}
```

The left trailing spaces of a statement is called the **indentation** of the statement:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
    TODO: left-right red arrow for indentation

    return 0;
}
```

It's a common programming style to use 4 spaces for indentation.

TODO ... Indentation.

0.6 Special characters

Hmmmm ... you don't see the `\n` printed out. In fact what's it for???

Try this:

```
#include <iostream>

int main()
{
    std::cout << "Hello, \nworld!\n";

    return 0;
}
```

And this

```
#include <iostream>

int main()
{
    std::cout << "Hel\nlo, \nworld!\n";

    return 0;
}
```

And finally this:

```
#include <iostream>

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

    return 0;
}
```

So you can think of the print cursor as jumping to the next line whenever a `\n` is encountered.

As a matter of fact you cannot separate the `\` from the `n`. `\n` is consider ***one*** single character. So technically I should write `'\n'`. `'\n'` is called the **newline character**.

Exercise 0.6.1. Write a program that prints this:

```
^  ^
0  0
 |
---
```

□

Try this:

```
#include <iostream>

int main()
{
    std::cout << "1\t2\tbuckle my shoe\n3\t4\t...\n";

    return 0;
}
```

The `'\t'` move the print cursor to the next tab position. `'\t'` is also a character. It's called the **tab character**.

Special characters like the above (the newline and the tab characters) are not printable and so we have to use some special notation to say "the newline character" or the "the tab character". Computer scientists decided (long time ago) to use \ to indicate a special character. These are called **escape characters**.

Now recall that `"` is used to mark the beginning and end of a string. What if you want to print something like this:

```
.....".....
```

Try this:

```
#include <iostream>

int main()
{
    std::cout << "....."....."; 

    return 0;
}
```

Doesn't work right? Do you see why?

Now try this:

```
#include <iostream>

int main()
{
    std::cout << "He shouted, \"42!\"\n";

    return 0;
}
```

So in a string, `\"` will let you print `"`.

`'\"'` is actually a character. There are two `'\"'` characters in the above string `"He shouted, \"42!\"\n"`.

Exercise 0.6.2. Write a program that prints this:

```
She said, "I would rather marry a pig."
```

☐

Exercise 0.6.3. Write a program that prints this: Here's a standard formula:

```
       2      2          2
(x + y)   =  x  + 2xy + y
```

There are two `' '` characters on each side of the `'='` character. ☐

Exercise 0.6.4. Write a program that prints this:

```
 d  3     2
-- x  = 3x
dx
```

☐

Exercise 0.6.5. How would you print the character `'`. Let me tell you that

```
std::cout << ''';
```

won't work. ☐

Exercise 0.6.6. Write a program that print the backslash character, i.e. `\`. ☐

Exercise 0.6.7. Write a program that prints this:

```
He said, "I am! I'm a pig! Really!"
```

□

Exercise 0.6.8. Write a program that prints this:

```
Get the Gold!
+-------------+
|             |
| --------+ | |
|         | | |
| +---+ | | | |
| |G  | | |   |
| +-+ +-+ | | |
| | |       | |
| | +-----+-+ |
|               <---
+-------------+
```

□

Exercise 0.6.9. Write a program that prints this:

```
#include <iostream>

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

    return 0;
}
```

□

The number of characters is a string is called **length** of the string. For instance the string

`"Hello, world!\n"`

has a length of 14.

Exercise 0.6.10. What is the number of characters (i.e., the length) of the following strings?

- `"columbia"`
- `""`
- `"columbia,mo"`
- `"columbia, mo"`
- `"ma ma mia!"`
- `"ma ma mia!\n"`
- `"ma\tma\tmia!\n\n"`

□

(You'll see much later that in every string, there's actually an extra secret character. This is however not included in the count of the length of the string. Don't worry about this for now. I will be coming back to this later.)

0.7 Printing More than One Strings or Characters

Try this

```
#include <iostream>

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

    return 0;
}
```

And this:

```
#include <iostream>

int main()
{
        std::cout << "Hello, " << "world!\n"
                  << "Spam, \n";

        return 0;
}
```

And this:

```
#include <iostream>

int main()
{
        std::cout << "Hello, " << "world!\n"
                  << "Ham, " << "eggs! \n";

        return 0;
}
```

And this:

```
#include <iostream>

int main()
{
        std::cout << "Hello, " << "world!\n"
                  << "Ham, " << "e" << "ggs! \n";

        return 0;
}
```

By the way you should try this:

```
#include <iostream>

int main()
{
    std::cout << "a" << "" << "b";

    return 0;
}
```

`""` is an **empty string** (also called a null string): it's a string with no characters.

Exercise 0.7.1. The output of the following program:

```
#include <iostream>

int main()
{
    std::cout << "ham", "and", "eggs"
              << "\n";

    return 0;
}
```

is

```
ham and eggs
```

True or false? (Verify with your C++ compiler.) ☐

Exercise 0.7.2. The output of the following program:

```
#include <iostream>

int main()
{
    std::cout << "ham" << "and" << "eggs"
              << "\n";

    return 0;
}
```

is

```
ham and eggs
```

True or false? (Verify with your C++ compiler.) ☐

Of course you can also print multiple characters. Try this:

```
#include <iostream>

int main()
{
    std::cout << 'H' << 'a' << 'm' << '\n';

    return 0;
}
```

Here's a tiring hello world program:

```
#include <iostream>

int main()
{
    std::cout << 'H' << 'e' << 'l' << 'l' << 'o'
              << ',' << ' '
              << 'w' << 'o' << 'r' << 'l' << 'd'
              << '!'
              << '\n';

    return 0;
}
```

You can also mix printing strings and characters in a single statement:

```
#include <iostream>

int main()
{
    std::cout << "Hel" << 'l' << "o, "
              << 'w' << "orld!\n";

    return 0;
}
```

0.8 Another Way to Go to the Next Line

Let's go back to our hello world program:

```
#include <iostream>

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

    return 0;
}
```

Run yours and make sure there are no errors.

First let's make a separate string out of the newline character:

```
#include <iostream>

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

    return 0;
}
```

Run it and make sure the effect is still the same.

Now do this:

```
#include <iostream>

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

    return 0;
}
```

Run it.

The `std::endl` acts like a newline character. It's actually more than that. We'll talk about this when we talk about files. Anyway for the time being just remember that `std::endl` forces a newline when you print it.

Exercise 0.8.1. Read this program and, without running it, write down the output in the grid provided below. Once you're done writing down the output, run the program and verify that you output from reading the program matches the output you get when running the program.

```
#include <iostream>

int main()
{
    std::cout << "Lives: 3" << std::endl
              << "Energy: 15000" << std::endl
              << "Score: 0" << std::endl;

    return 0;
}
```

Output (one per character):

0.9 Multiple Statements

Exercise 0.9.1. Run this program

```
#include <iostream>

int main()
{
    std::cout << "1!\n";
    std::cout << "2!\n";

    return 0;
}
```

This tells you that you can have **more than one statement** in your program. (Actually the `return 0;` is also a statement, but we'll ignore that for now.)

Furthermore C++ executes from **top to bottom** (at least right now!)

Exercise 0.9.2. The following program has three print statements:

```
#include <iostream>

int main()
{
    std::cout << "Hello World! ";
    std::cout << "I'm the queen of England. ";
    std::cout << "I have 3 arms.";

    return 0;
}
```

Run it. Rewrite it so that it has only one print statement. □

Exercise 0.9.3. True or False? The output of this program (when you run it of course)

```
#include <iostream>

int main()
{
    std::cout << "this is the last line\n";
    std::cout << "this is the second line\n";
    std::cout << "this is the first line\n";

    return 0;
}
```

is

```
this is the first line
this is the second line
this is the last line
```

□

Exercise 0.9.4. Continuing with the previous program, rewrite it so that each sentence is printed on a separate line; use only one statement. □

0.10 The `using namespace std` business

Instead of our hello world program:

```
#include <iostream>

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

    return 0;
}
```

in many books you will see this:

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, world!\n";

    return 0;
}
```

All the

```
using namespace std;
```

does is that after it, instead of

```
std::cout
```

you can write

```
cout
```

i.e. without the

```
std::
```

That's all. I'll talk about "namespaces" later (actually much, much, much later ... in CISS245). So don't worry too much about it.

Exercise 0.10.1. Does the following program work?

```
#include <iostream>
using namespace std;

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

    return 0;
}
```

□

So not only can you replace `std::cout` with cout after doing the

```
    using namespace std;
```

you can also replace `std::endl` with `endl`.

0.11 Summary

A **statement** is something that will cause your computer to perform some operation(s). In C++ statements must terminate with a semi-colon.

C++ is case-sensitive.

C++ ignores whitespace (between tokens).

Something that looks like `"Hello, World!\n"` (i.e. characters within double quotes) is called a **string** or a **C-string**.

A **string** is either empty (the null string), i.e. `""`, or it is made up of characters. For instance the first character of the string

```
"Hello, World!\n"
```

is `'H'`.

There are special characters: `'\n'` causes the cursor to jump to a new line while `'\t'` moves to the next tab position, `'\"'` is the double-quote character, and `'\''` is the single-quote character. The backslash character is `'\\'`. These are called **escape characters**.

To print a string, you execute the following statement:

```
std::cout << [some string];
```

You can print more than one string in a single print statement:

```
std::cout << [string1] << [string2]
          << [string3] << [string4];
```

Printing one or more characters is the same:

```
std::cout << [some character];
std::cout << [character1] << [character2]
          << [character3] << [character4];
```

You can mix print strings and characters. For instance

```
std::cout << [some character]
          << [some string];
```

or

```
std::cout << [some string]
          << [some character];
```

You can force a newline by printing `std::endl`:

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

If your program has two or more statements like this:

```
[statement 1];
[statement 2];
[statement 3];
```

then C++ executes top to bottom (for now).

0.12 Exercises

1. Write a C++ program that produces the following output:

```
"Prediction is very difficult,
especially about the future."
    Niels Bohr
    Danish physicist (1885 - 1962)
```

Use one print statement.

2. True or false: The output of

```
#include <iostream>

int main[]
{
    std::cout << "To be ..."
              << "or not to be ..."
              << "That" << "is the question."
              << std::endl

    return 0
}
```

is

```
To be ...
or not to be ...
That is the question.
```

Verify with your C++ compiler.

3. Find and correct all the error in this program (without using your C++ compiler):

```
#include <<iostream>>

int main[]
{
    std::cout >> "Hello!/n"
    std::cout >> "I'm the king of Spain./n"
    std::cout >> "And I have 3 arms." >> std::end

    return 0
}
```

The expected output is

```
Hello!
I'm the king of Spain.
And I have 3 arms.
```

When you're done, verify your correction with your C++ compiler. If your program does not compile, continue correcting the program until the program is error-free and runs correctly.

4. Will this program get an A from the instructor?

```
#include <iostream>

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

    return 0;
}
```

Why? What about this:

```
#include <iostream>

int main()
{


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


    return 0;
}
```

5. What is the length of the following string:

```
"Ablee, \tAblee, \tAblee, \t... That's all folks!!!\n"
```

6. A string with length 0 is called an empty string or a ___________ string.

7. All C++ statements must end with a ______________________.

Chapter 1

Integer Values

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.

1.1 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()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

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

1.2 Printing integers

An **integer** is just 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.

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!!!

1.3 Integer operators: +, -, and *

Now you might say, "Isn't this ..."

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

Now you might say, "Isn't this ..."

```
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!!!

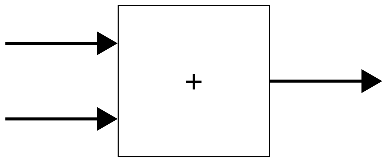
Try this:

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

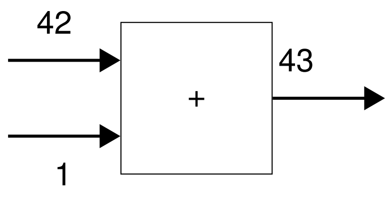
So far we see that C++ understands three **operators**: +, –, and *.

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:



n 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 5 of 34 **integer + operator**.

The story is very similar for operator - and *

1.4 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.

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.

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.

irst of all you see that % is an operator. What does the operator % do? Answer: % is called the **mod** operator.

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:

134 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?

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.

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.

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.

1.5 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++.**

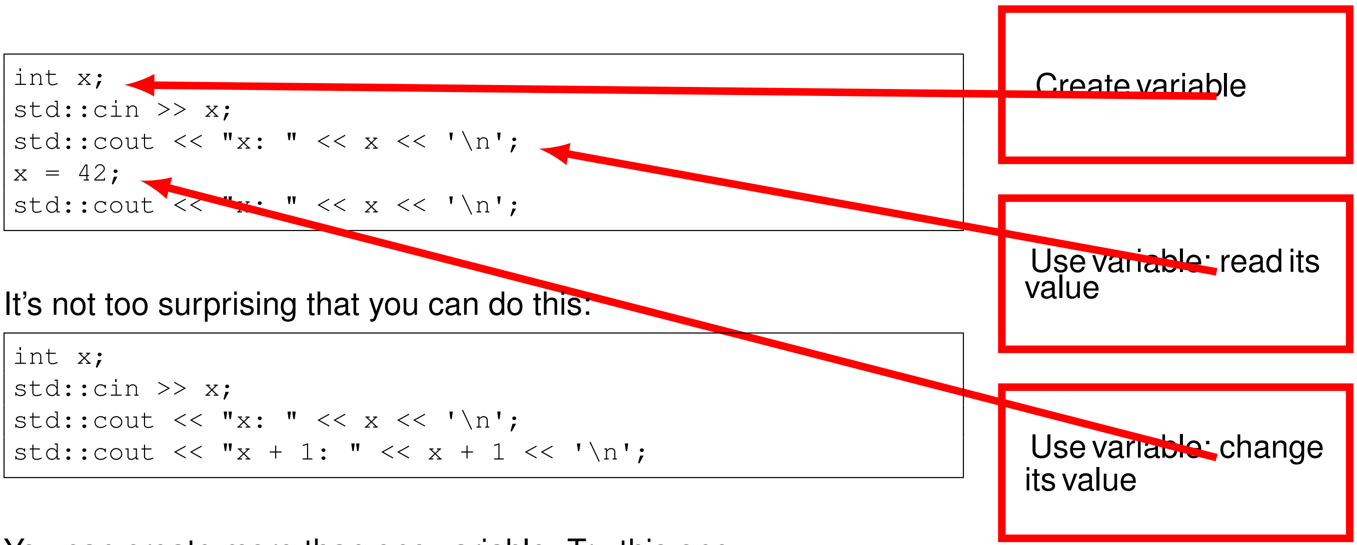
1.6 Variables: input and output

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.

variable – it holds a value and the value can vary. x is an integer 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** 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';
```

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.

1.7 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

$$\frac{61}{3} = 20\frac{1}{3}$$

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.

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,

$$100 = 4x25$$

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:

$$100 = 2^2x5^2$$

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.

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:

$$42 = 2x21$$

Of course 3 divides 42 as well since

$$42 = 3x14$$

But in order to say that 42 is not a prime, you only need to find one 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
- is not a divisor of 7
- is not a divisor of 7
- is not a divisor of 7
- 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

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).

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 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?!?

1.8 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.

1.9 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.

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):

$$1 \times 1000 + 3 \times 100 + 5 \times 10 + 8 \times 1$$

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?)

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,

$$3 \times 100 = 300$$

i.e., multiplication by 100 just adds 2 zeroes to the given number.

Here's another example:

$$8 \times 10000 = 80000$$

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

$$867 \times 100000 = 86700000$$

1.10 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?

1.11 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.

```
\li The left + is applied to 2 and 3.
\li 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:

$$\begin{aligned}&(2+3)+1 \\ &= 5+1 \\ &= 6\end{aligned} \qquad\qquad \begin{aligned}&2+(3+1) \\ &= 2+4 \\ &= 6\end{aligned}$$

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

$$\begin{aligned}&(2-3)-1 \\ &=-1-1 \\ &=-2\end{aligned} \qquad\qquad \begin{aligned}&2-(3-1) \\ &=2-2 \\ &=0\end{aligned}$$

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.

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:

$$\begin{aligned}&1 + 3–\underline{2 * 6}/3 * 4 + 6\\&= 1 + 3–12/3 * 4 + 6 &&i.e., 2 * 6 = 12\end{aligned}$$

The next in the second priority operators is here:

$$1 + 3–\underline{12/3} * 4 + 6$$

Let's do that:

$$\begin{aligned}&1 + 3–\underline{12/3} * 4 + 6\\&= 1 + 3–4 * 4 + 6 &&i.e., 12 * 3 = 4\end{aligned}$$

Got it?

And the last evaluation for the second priority operators give us this:

$$\begin{aligned}&1 + 3–\underline{4 * 4} + 6\\&= 1 + 3–16 + 6 &&i.e., 4 * 4 = 16\end{aligned}$$

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:

$$\begin{aligned}
&\underline{1+3}\text{–}16+6 && \\
&= \underline{4\text{–}16}+6 && i.e., 1+3=4 \\
&= \underline{\text{–}12+6} && i.e., 4\text{–}16=-12 \\
&= \text{–}6 && i.e., -12+6=-12
\end{aligned}$$

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:

$$\begin{aligned}
&1+3\text{–}\underline{2*6}/3*4+6 && \\
&= 1+3\text{–}\underline{12/3}*4+6 && i.e., 2*6=12 \\
&= 1+3\text{–}\underline{4*4}+6 && i.e., 12/3=4 \\
&= \underline{1+3}\text{–}16+6 && i.e., 4*4=16 \\
&= \underline{4\text{–}16}+6 && i.e., 1+3=4 \\
&= \underline{\text{–}12+6} && i.e., 4\text{–}16=-12 \\
&= \text{–}6 && i.e., -12+6=-12
\end{aligned}$$

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.

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

$$3+1*4\text{–}4*3*2/(2+6*2\text{–}1)+6*7$$

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:

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

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:

$$\begin{aligned}
&3 + 1 * 4–4 * 3 * 2/(\underline{2 + 12}–1) + 6 * 7 \\
&= 3 + 1 * 4–4 * 3 * 2/(\underline{14–1}) + 6 * 7 && i.e., 2 + 12 = 14 \\
&= 3 + 1 * 4–4 * 3 * 2/(13) + 6 * 7 && i.e., 14–1 = 13 \\
&= 3 + 1 * 4–4 * 3 * 2/13 + 6 * 7 && i.e., (13) = 13
\end{aligned}$$

(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:

$$\begin{aligned}
&3 + 1 * 4–4 * 3 * 2/13 + 6 * 7 \\
&= 3 + 4–4 * 3 * 2/13 + 6 * 7 && i.e., 1 * 4 = 4 \\
&= 3 + 4–12 * 2/13 + 6 * 7 && i.e., 4 * 3 = 12 \\
&= 3 + 4–24/13 + 6 * 7 && i.e., 12 * 2 = 24 \\
&= 3 + 4–1 + 6 * 7 && i.e., 24/13 = 1 \\
&= 3 + 4–1 + 42 && i.e., 6 * 7 = 42
\end{aligned}$$

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 –.

$$\begin{aligned}
&3 + 4–1 + 42 \\
&= 7–1 + 42 && i.e., 3 + 4 = 7 \\
&= 6 + 42 && i.e., 7–1 = 6 \\
&= 48 && i.e., 6 + 42 = 48
\end{aligned}$$

and we're done. Study every single step of the above computation.

f you have an expression with <u>two</u> 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.

$$\begin{aligned}
2 * (3 + 5) - 3 * (5 + 3 * 2) &= 2 * (8) - 3 * (5 + 3 * 2) && i.e., 3 + 5 = 8 \\
&= 2 * 8 - 3 * (5 + 3 * 2) && i.e., (8) = 8 \\
&= 2 * 8 - 3 * (5 + 6) && i.e., 3 * 2 = 6 \\
&= 2 * 8 - 3 * (11) && i.e., 5 + 6 = 11 \\
&= 2 * 8 - 3 * 11 && i.e., (11) = 11 \\
&= 16 - 3 * 11 && i.e., 2 * 8 = 16 \\
&= 16 - 33 && i.e., 3 * 11 \\
&= -17 && i.e., 16 - 33 = -17
\end{aligned}$$

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.

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

1.12 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:

-(-3) is 5

$$+2$$

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

+5 is 5

$$-2$$

So the symbol + and – have two different meanings.

Now we need to modify the chart of precedence rules:

$()$	first priority
$*, /, \%$	second priority
$+, -$	third priority

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).

1.13 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
$*, /, \%$	second priority
$+, -$	third priority

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

$$1 + 2 – 3 + 4 \quad \text{is} \quad ((1 + 2) – 3) + 4$$

and

$$- + - 1 \quad \text{is} \quad -(+(-1))$$

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:

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