LESS CONFUSING C++

Cracking CBSE's 11th grade C++, made uncomplicated.

Namanyay Goel

Less Confusing C++ Part 1

Cracking CBSE's C++ made uncomplicated

Namanyay Goel

This book is for sale at http://leanpub.com/lessconfusingcpp

This version was published on 2017-08-16

Tweet This Book!

Please help Namanyay Goel by spreading the word about this book on Twitter!

The suggested hashtag for this book is #lessconfusingcpp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#lessconfusingcpp

Contents

1. How to use this book

Less Confusing C++ aims to create a *better understanding of C++* and help you *score more marks* in 11<sup>th</sup> grade's IT. I **strongly suggest** that you read every bit of this book, even if some parts seem superflous to you–they probably are required to gain easier understanding of future topics.

The book assumes at least basic knowledge of C++ taught to you in school. I've tried being extensive, but it is not always possible.

It would be best if you download Turbo C++ on your own computer, and keep it in front of you while reading this book. You can download the 'school' version of the compiler by searching "Class 11 CBSE Turbo C++" on Google. Write and try out the code samples given, tinker with the code and make mistakes. Remember, your compiler is the best teacher.

> There often is some text that is required for better explanation, but isn't a part of the syllabus. That is boxed like it is done here.

CBSE answers are given in this format (with a book on the left). They often refer to previous definitions and code.

Finally, the book is targeted towards C++ as taught to you in your school. New, modern versions (11.x, instead of 3.x or 4.x seen in schools) of C++, written using better integrated development environments (Code::Blocks, Visual C++), will make you realize that C++ is a much more beautiful language than you were led to believe.

1.1 Feedback

Send over any feedback to mail [at] namanyayg.com

Some terms to be familiar with

- **Block**: A block of code is any code contained within braces (`{}`).
- **Whitespace**: A space, tab, or a line break is whitespace. Used for clarity and practical needs.
- **Pseudocode**: Literally ‘fake code’. A human description of a program. If you try to run it, it won’t. Used for simpler explanations of complex programs throughout the book.

2. Walkthrough of a simple program

We'll be writing a simple C++ program: One that outputs 'Hello World'. Excited yet?

Here's all the code:

Hello World!:

```cpp
#include <iostream.h>

void main()
{
    cout << "Hello World";
}
```

I'm pretty sure many of you readers will be familiar with the above code. Nonetheless, I'll be highlighting each part of the code and explaining things.

1. `#include <iostream.h>`: This is the example of a *preprocesser directive*. In this case, it is an *include statement*.
2. `void main() {...}`: This is the *entry point* of every C++ program. It is also a *function declaration*, but that will be covered later.
3. `cout << "Hello World";`: It is an *output statement* using `<<`, an *insertion operator*.

2.1 Pre-processor Directive

A pre-processor directive is a *special instruction* sent to the compiler, before the actual execution of the program. It starts with the hash symbol (`#`).

This is the definition you'll get in your school. But what does the 'special instruction' do?

2.1.1 Including

This is the most common type of preprocessor directive you'll see. Examples:

Some include statements:

```
#include <iostream.h>
#include <iomanip.h>
```

What this does is **include the contents of the specified file in your program**.

The `*.h` files you see are called *header files*. They define functions and variables that make it easier for you to write programs.

For example, `iostream.h` defines `cout` and `cin`, that allow you to give console **out**put and take console **in**put. `iomanip.h` defines `setw()` that allows you to set a width to console outputs.

However, as many people believe in the start of the term, **preprocessor directives are NOT only include statements**, there are others too. Which leads us too...

2.1.2 Defining constants

Constant definition:

```
#define MAX 10000
```

Note the lack of semi colons (`;`*) or equality symbols (*`=`*).*

This defines a **constant** `MAX` to be of value `10000`. Changing `MAX` anywhere in the code will give you an error (*i.e.,* `MAX = 5` will give you an compiler error).

Therefore, the correct answer to the CBSE question is the above definition + one example of an 'include' statement and one example of the 'define' statement, both given above.

2.2 void main()

`main()` is the only function recognized by every C++ program. The compiler checks for the `main()` *function*, and executes all it's contents. The compiler does nothing without the `main()` function. The word `void` before `main()` indicates that the function *does not return* any data.

`void main()`:

```c
...
void main()
{
    ...
}
```

Functions

What is a function, you asked? Put simply a function is a *named set of instructions* that you can refer to again and again. A function can take *arguments* (in parentheses) which it can then operate upon.

`main()` is such a function, it's named `main` and takes no arguments (Hence the empty parentheses `()`).

Another example of a function is `pow()`, included in the `<math.h>` library. It takes two integers as *arguments*, one for the base and other the exponent, and *returns* base^exponent. Example, `pow(2, 3)` will give you `8`. Here, the first *argument* was "2" being the base, and `3` being the exponent.

When a function is called, the control stops and starts running at the first line of the function. Functions allow for 'flow control', and separate the program into a number of sub-steps.

Entry Point

The `main()` function is referred to the 'entry point'. This is because `main()` is the place where the compiler looks for all instructions, *i.e.,* it is the *point* where the compiler *enters*.

The `void` part of `void main()` simply says that `main()` will not *return* any value. This is contrary to `pow()` (for example) that returns an `int` value. `void` is one of the *fundamental datatypes*. (Explained in more detail later).

2.3 Console Ouput

"Console Out", or `cout` statements are used to print text on a user's screen.

The above example would output "Hello World!" to your console (the console can be viewed by pressing `ALT + F5`).

The `cout` stream is defined in the `iostream.h` header file (**I**nput **O**utput Stream or I/O Stream). Hence, you'll see this header file included at the start of almost every program, as it is crucial to taking user input or showing output to the user.

Streams

Streams are sequences of data made available over time, like a conveyer belt. Input and output devices are called 'streams', since unlimited data can potentially be created or taken over infinite time. `cout` is an output stream which sends data to the screen.

The two opening triangle brackets (`<<`) make up the *insertion operator*. This *inserts* from right to left (the direction of the arrows). This is also colloquially known as the 'output' operator.

`cout`:

```
...
cout << "Hello World";
...
```

3. Introduction to Identifiers

In the last chapter, we briefly discussed about the possibility of storing user input.

Consider when you want to change the program depending on what the user enters.

Or, consider if you want to store a oft-repeated value so you save time when you need to repeat it.

In both the cases, the solution is creating and using *variables* or *identifiers*.

3.1 A quick example

Let's say you wish to write a program that takes a integer input from the user and returns the cube of the number. In pseudocode, that might look like:

Pseudocode Cubing:

```
#include <iostream.h>
void main()
{
    take user input as integer, and store it in 'number';
    cout << number * number * number;
}
```

We'll need to figure out how to write `take user input as integer, and store it in 'number'` so that the computer understands it.

We can breakdown the statement into **2** simple parts:

1. *Declare* a identifier which stores user input.
2. Take *input* from the user and assign it to the identifier

3.1.1 Declaration

We'll first need to figure out what *type* of input will the user enter. There are many types you can have (more of which will be explained later), but currently we simply want a interger, or, we need `int` type input.

Secondly, we'll need to think of a 'word' to call the input with. Just like you don't say *Canis lupus familiaris*, but 'dog'; similarly, the 'word' will help us to point at the data in a more human manner. This 'word' is formally called the *identifier* or the *variable name*.

(Note: I will be using the terms identifier and variable name interchangeably ahead. They both, however, mean 'word')

3.1.1.1 Naming Rules

The compiler has some strict rules in naming the variable. These are:

- An identifier can only contain alphanumeric (alphabets and digits) and the underscore characters.
- An identifier *must* start with either an underscore or an alphabet.
- Identifiers are case sensitive, *i.e.,* `FOO` and `foo` are treated as separate.
- Maximum length of a variable name can be 32 characters.

3.1.1.2 Naming Conventions

> "There are only two hard things in Computer Science: cache invalidation and naming things." – *Phil Karlton*

Whenever we create a new *variable name*, it is helpful to follow some conventions so that our names are consistent and simple to write. Summarized, identifiers should be:

- Short.
- Meaningful.
- Self-expressive.

These will be explained in detail in the next chapter.

3.1.1.3 Code

To put all this together, let's term the identifier to be `number`. We've made it be of the `int` datatype already. So to declare the variable, write:

Declaring a variable:

```
...
int number;
...
```

3.1.2 Console Input and assignment

In the last chapter we learnt that `cout`, which allows for console output, is defined in `<iostream.h>` header file.

Similarly, for taking input, `<iostream.h>` defines the `cin` (Console Input) stream which enables the user to enter input.

`cin` uses the *extraction operator*. It looks like the mirror image of the insertion operator, *i.e.,* `>>`, and works in a opposite manner too. `>>` *extracts* data from left to right.

For this example, we need to extract user input and place the value in the `number` variable.

Using `cin`:

```cpp
...
int number;
cin >> number;
```

We can also say that we have *assigned* `number` to be the user's input. (More about assignment later).

3.2 Theory

3.2.1 Datatypes

Data can be of many types. You might want to manipulate only integers, or perhaps you want high precision in decimals. You might also need to store a symbol or a character.

Datatypes become important here. Since there can be a variety of data, there should be a variety of ways to handle, store, and manipulate such data.

3.2.1.1 Fundamental Datatypes

C++ has five 'fundamental' datatypes.

Type	Description	Size (Bytes)
`int`	This represents integers without decimal points. If you force decimal points (through user input or perhaps by division), values are rounded down.	2
`char`	This represents character values. Can store any character such as `a` or `&`. This is a subset of the `int` datatype and converts integers to characters through ASCII (American Standard Code for Information Interchange).	1
`float`	This represents real numbers with decimals. Allows up to 6 digits after the decimal point.	6
`double`	This too represents floating point numbers, but allows for a larger range and a higher precision (15 digits after decimal point)	15
`void`	This represents an empty set of values. It is used mainly in functions to specify that no data will be returned, example `void main()`. Variables of type `void` can't be declared	-

Sizes and range

1 byte contains 8 bits. Each bit can be either 1 or 0. Hence, each byte can store upto 2^8 unique values.

The `int` datatype has a size of 2 bytes. Therefore, it can store a maximum of 2^{16} unique values.

Binary to decimal

Let's consider a byte, `00101010`.

Binary is a base two number system, and binary (Base 2) can be converted to decimal (Base 10) using the following method.

To convert this binary string to number, we multiply the units place by 2^0, tens place by 2^1, and so on.

The conversion for the above string would thus be:

`00101010`

- 1st place: 0 multiplied by 2^0 = 0
- 2nd place: 1 multiplied by 2^1 = 2
- 3rd place: 0 multiplied by 2^2 = 0
- 4th place: 1 multiplied by 2^3 = 8
- 5th place: 0 multiplied by 2^4 = 0
- 6th place: 1 multiplied by 2^5 = 32
- 7th place: 0 multiplied by 2^6 = 0
- 8th place: 0 multiplied by 2^7 = 0

The 7th and 8th places will be 0, since 0 * x = 0.

Summing up the numbers, 2 + 8 + 32, we get "42"

There exist negative integers as well, so `int` needs to store negative values. One bit is assigned to storing the 'sign', if it is `1` then the number is positive, if it is `0` the number is negative. This leaves 15 bits for the actual number, or 2^{15} unique values.

The range for the `int` datatype thus is [-2^{15}, (2^{15}-1)], which in numbers is [−32,768, +32,767]. `int` goes up only till +32,767, not +32,768 because 0 is one unique value as well.

Simiarly, `char` can have 2^8 unique values.

Here are the sizes ranges of the above datatypes:

Type	Size (Bytes)	Range
`int`	2	[-32167, 32168]
`char`	1	[-128, 127]
`float`	6	[(3.4×10^{-38}), ($3.4 * 10^{38}$)]
`double`	15	[(1.7×10^{-308}), (1.7×10^{308})]
`void`	-	-

Thus, the definition of any one datatype is it's description, size, and range

3.2.1.2 Modifiers

In C++, you can add **datatype modifiers** on `char`, `int`, and `double` data types. Modifiers precede the data type, and allow to change the base type to better fit with the requirements of the situation. All basic data types, except `void`, can have modifiers preceding them.

There are four modifiers that can be used in C++.

- `unsigned`
- `signed`
- `long`
- `short`

Unsigned

The unsigned data type is used when you don't want the data to allow negative inputs, *i.e.,* have only positive values. This modifier can only be used for `int` and `char` type values.

The size occupied by the variable remains the same, but, the range is 'shifted'.

Let's take up the example of using `unsigned` on the `int` datatype. A good use case would be when you're counting population—this can in no way go in negative.

Using the unsigned modifier:

```
unsigned int population;
```

As we've seen earlier, the range of `int` is [-2^{15}, (2^{15}-1)]. When you modify `int` using `unsigned`, the range 'shifts' to [0, 2^{16}-1], or, [0, 65535]. It still occupies 2 bytes.

By default, just writing `unsigned b` will make `b` of the integer type, not the character type.

Signed

The `signed` modifier is the 'default' value of the `int` and `char` datatype. This modifier, like `unsigned`, can only be used on `int` and `char`. Again, the size remains the same. The signed modifier allows having negative values.

Using the signed modifier:

```
signed int x;
```

Long

The `long` modifier increasing the size, and thus the range, of the data type it is used with. `long` can only be used for `int` and `double` data types.

When used with `int`, the data type gets **4 bytes** of memory. The range increases to [2^{31}, 2^{31}-1], or, [-2,147,483,648, 2,147,483,647].

`long double` occupies **10 bytes** of memory.

Again, the default datatype when the `long` modifier is used is `int`, so `long int a;` and `long a;` mean the same thing.

Short

`short` is the default way data types are stored. `short int a;` is the same as `int a;`, each using 2 bytes of storage and ranging from [-2^{15}, 2^{15}-1].

For the definitions of modifiers, always specify:

- A short description of their working.
- The range of the values.
- One example showing the syntax.

Modifiers are useful because it's best that you choose the most efficient data type you require. If you need values of up till 40000, the `double` datatype is overkill while `long int` allows you to get a more optimized and faster program.

3.2.2 Variables in memory

We've learnt that variables can be used to store data of different types, but what actually is a variable?

A variable is simply a reference to a *memory location*. Memory is usually represented as a set of 'blocks'.

Memory

Variables are stored in the RAM (Random Access Memory). This is also called as the 'primary storage' or simply 'memory'.

The advantage of storing data in RAM is that it's very fast when compared to 'secondary storage' or 'storage', usually a hard disk. However, RAM's disadvantages are that it is temporary and has lesser space than the secondary storage. Newer computers have around 4 - 8 GB of RAM, while they have 500 - 1000 GB of storage.

When C++ was developed, the average computer had around 128 KB and around 10 MB of storage.

Here's a simple representation of 8 byte memory that shows variable storage.

A crude representation of memory:

```
 ___
|___| 1
|___| 2
|___| 3
|___| 4
|___| 5
|___| 6
|___| 7
```

Let’s say we create a variable `letter` of type `char` and store the character `a` in it. C++ will link a ‘block’ to the variable `letter`. Let’s assume it assigns block number ‘5’ to `letter`. Then, a representation of the memory could be:

Variables in memory:

```
 ___
|___| 1
|___| 2
|___| 3
|___| 4
|_a_| 5 = letter
|___| 6
|___| 7
```

A variable is a name given to a memory location where data of a particular datatype is stored.

3.2.3 Putting it all together

A variable is a name given to a memory location where some data is stored. It can be of many data types, and the data type is decided by the data stored in it. For example, integers are stored in the `int` data type, characters are stored in the `char` data type.

4. Writing Beautiful Code

Thus spake the master programmer: “A well-written program is its own heaven; a poorly-written program is its own hell.” *–The Tao of Programming*

Programs are for humans, not for computers (otherwise we all would be writing in `1`s and `0`s). Every human reading your code should have a easy time reading what you’ve written. We’ll establish a few guidelines to be followed when writing code to make your code pleasant to read, *i.e.,* we’ll write a simple *style guide.*

Some parts of the style guide are compulsory, you **will** lose marks if you don’t follow them. Others are for prettier and better code. Following all is Good Idea^{TM}, nonetheless.

Ultimately, you can write C++ in any way (without spaces and tabs), completely disregarding this style guide, if you wish–it all is valid code. However, it will be *extremely* confusing to read and you *will* make silly errors. Following a style guide *(like this one)* is thus a good idea. Treat the ‘guides’ as ‘rules’, and you’ll breeze through writing good code.

4.1 Style Guide

4.1.1 Spacing

Be generous with spaces. Use them a lot, use them everywhere.

Examples of incorrect spacing:

```cpp
int i=0;
if (i>5)
{
    ...
}
i+=5;
```

would be wrong.

Better spacing:

```cpp
int i = 0;
if ( i > 5 )
{
    ...
}
i += 5;
```

is better.

Keep in mind, though, that sometimes you want *not* to have spaces. For example, `i ++;` will give you an error - here `i++;` is correct.

4.1.2 Indentation

We'll be leaving a tab space in the start of a new block. This allows for easier readability at a glance, since you know where a new block starts.

Tab Size

Turbo C++, by default, sets indentation level to '8 spaces'. '4 spaces' is much more reasonable of a length and also allows for easier nesting. Change this by going to `Options > Environment > Editor > Tab Size` and set the value to '4'.

Unclear indentation:

```cpp
void main()
{
cout << "help";
}
```

isn't correct, write:

Better indentation:

```cpp
void main()
{
    cout << "help";
}
```

instead.

Never write something like this:

Do not try at home:

```
void main()
{
int a,b,c;
  cin >> a;
cout << a;
    cin >> b;
}
```

Shudders.

4.1.3 Round Brackets `()`

AKA brackets, parentheses, or parens

Parens are seen in conditionals (`if`/`switch`) and loops (`for`/`while`) in the first term.

Leave spaces after the beginning paren, and before the ending paren.

Incorrect paren spacing:

```
for (int i = 0; i < 10; i++)
{
    ...
}
```

is incorrect, write:

Better paren spacing:

```
for ( int i = 0; i < 10; i++ )
{

}
```

instead.

4.1.4 Curly Brackets `{}`

AKA braces or flower brackets

Braces everywhere should be written after leaving a *line break*.

Incorrect braces line breaks:

```
void main() {
    cout << "Hello World";
}
```

is bad, write the following instead:

Proper braces spacing:

```
void main()
{
    cout << "Hello World";
}
```

Similarly,

More incorrect braces spacing:

```
for ( int i = 0; i < 10; i++ ) {
    cout << i;
}
```

isn't a good idea, try writing:

Better braces spacing:

```
for ( int i = 0; i < 10; i++ )
{
    cout << i;
}
```

4.1.5 Comments

Comments are 'notes' written for the sake of the programmer for clarity of code, rather than the compiler. They are completely ignored by all compilers.

Comments might look futile and a waste of time at the first glance–however, they are extremely useful for yourself and your teachers, and can net you better marks in practicals ;).

> "Though a program be but three lines long, someday it will have to be maintained."
> –*The Tao of Programming*

There are two types of comments:

4.1.5.1 Block comments or Multi-line comments:

Block or multi-line comments start with `/*` and end with `*/`, and can span multiple lines. Useful for explaining code in detail.

Examples of block comments:

```
/*
Turns any number even:
(num % 0) evaluates to 1 if odd, and 0 if even.
if the number is odd, it adds 1 to itself. Otherwise,
remains the same.
*/
num += (num % 0);
```

It is recommended that block comments span reasonable line length, and their width is kept small through the use of manual line breaks by pressing the ‘enter’ key.

4.1.5.2 End-of-line comments:

End-of-line comments are used to provide short explanations to code. They begin with the `//` (double slash) sequence, and the complete line from the double slash is ignored by the compiler.

Examples of end-of-line comments:

```
for ( int i = 0; i < n; i++ ) // To loop through user input and iterate sum
{
...
}
x = float(pow(a, 3))/float(pow(b, 2)); // x = a^3/b^2
```

While writing comments are encouraged, use them *only* for difficult to comprehend blocks of code. Writing comments on obvious statements is a waste of time. Comments like those below are useless.

Examples of silly comments:

```
int i = 5; // Assigns '5' to i.
i = i / 2; // Divides i by 2
```

4.1.6 Variable naming

Variables should always have **meaningful and short names**. They should be as short as you can make them, but not shorter than what is required.

Poor variable naming:

```
int m1, m2, m3;
cout << "Enter marks: ";
cin >> m1 >> m2 >> m3;
```

`m1`, `m2`, `m3` are short names, yes. They might even make sense once you read the second line ("Enter marks"). But they aren't good variable names–a good variable name makes sense on the first read.

Better variable naming:

```
int marks1, marks2, marks3;
...
```

Another example would be of a 'flag' variable. **Flags** are used to store a characteristic, example, to store whether the given number is composite or not.

You would have been taught to create the flag variable with a name of `flag` or `f`. Both are poor ideas. Instead, expounding on my example above; if you wish to check primality of a number, use a variable such as `iscomposite` to store the characteristic.

Pseudocode demonstrating use of flags:

```
int num, iscomposite = 1;
cout << "Enter number";
cin >> num;

if ( num is prime )
{
    iscomposite = 0;
}

if ( iscomposite )
{
    cout << "You have entered a composite number!";
}
else
{
    cout << "The number you've entered is prime!";
}
```

Secondly, as a rule, identifiers should be in lowercase and should never start with an underscore.

5. True and False

One of the core features of each language is it's handling of 'true' and 'false'. You'll see this used *a lot* in the first term of 11th–in conditionals, in loops, everywhere.

Computers only know `1`s and `0`s. For them, true is anything not `0` and false is `0`. That's it.

Just keep this in mind, **false is `0` and true is anything that is not a `0`**.

We already know the 'not' sign, that is the exclamation mark `!`.

True and false:

```
false is 0
true is !0
true is 1
```

5.1 Relational expressions and logical operators

A relational operator is one that describes a relation between two values. Greater than, equal to, less than or equal to, etc - all describe a relationship between two numbers.

In C++, we have relational operators that form *relational expressions* when used. Examples of relational operators are equality (==), inequality (`!=`), less or greater than (<, >), less than or equal to or greater than and equal to (<=, >=).

A relational expression is two or more expressions whose values are compared according to the relational operator. Examples: `5 > 3`, `(5 > 3) > ( 4 < 2)`, etc.

Similarly, a logical expression is one which evaluates values according to logical operators such as AND (&&), OR (`||`), NOT (`!`)

5.2 Operations

Now, performing the common operations on each:

Operations on true and false:

```
// AND (&&)
cout << (1 && 1); // 1
cout << (1 && 0); // 0
cout << (0 && 1); // 0
cout << (0 && 0); // 0

cout << (1 && 5); // 1
cout << (0 && -1); // 0
cout << (-1 && -1); //-1

// OR (||)
cout << (1 || 1); // 1
cout << (1 || 0); // 1
cout << (0 || 1); // 1
cout << (0 || 0); // 0;

cout << (5 || 0); // 1
cout << (-1 || -1); // 1

// EQUALITY (==)
cout << (1 == 1); // 1
cout << (1 == 0); // 0
cout << (0 == 1); // 0
cout << (0 == 0); // 1

cout << (5 == 5); // 1
cout << (0 == -1); //0

// INEQUALITY (!=)
cout << (1 != 1); // 0
cout << (1 != 0); // 1
cout << (0 != 1); // 1
cout << (0 != 0); // 0

cout << (5 != 0); // 1
cout << (5 != 5); // 0
```

Figure a pattern in the above? (Hint: Consider the first four statements after each main comment).

You would have done a similar ‘truth table’ in your school, with 1 and 0 being replaced by T and F, respectively. Now you know what it means.

Expressions

An expression is a statement that results in a 1 (true) or 0 (false). Examples, `1==1` or `!0` or `a > 5`.

5.3 Applications

Consider, a simple conditional statement.

A simple conditional:

```
int a;
cin >> a;

if ( a == 5 )
{
    cout << "Hey there!";
}
else
{
    cout << "Bye!";
}
```

If you input `a` to be 5, the *expression* `( a == 5 )` will evaluate to be 1. The conditional is thus reduced to:

Reduced conditional:

```
if ( 1 )
{
    cout << "Hey there!";
}
else
{
    cout << "Bye!";
}
```

And as we know, this will result in the block inside `if` to run. We will see “Hey There!” in the console.

If `a` is anything else except 5, we will see “Bye!” in the console.

Thus, **reducing expressions** makes conditionals and loops easier to understand.

5.4 Order of evaluation

Just like BODMAS (Brackets Division Multiplication Addition Subtraction) tells what operator to use first in a mathematical expression, all expressions in C++ are evaluated in the order given below.

1. Brackets/Paren and post increment/decrement (`()`, `n++`, `n--`)
2. Pre increment/decrement (`++n`, `--n`)
3. Multiplicative (`* / %`)
4. Additive (`+ -`)
5. Relational (`< > <= >= ==`)
6. Equality (=)
7. AND (&&)
8. OR (||)
9. Ternary operator (`? :`)

This is also termed as *precedence*, which tells which operators are given more 'importance'.

Note: Don't worry if you don't understand some of the terms in the order of evaluation. They'll be discussed later

5.4.1 Questions:

Evaluate the following

1. `( 5 > 3 )`
2. `( 5 > 3 && 1 < 2 )`
3. `( (2 > 3 || 1 < 2) && (3 > 2) )`
4. `( 2 > 3 || 1 < 2 && 3 > 2 )`
5. `( 7 >= 25 || 5 + 5 > 9 && 2 == 3 )`

5.4.2 Answers:

1. `( 5 > 3 )`

This simply evaluates to `1`.

2. `( 5 > 3 && 1 < 2 )`

The expression needs to be *reduced*. The order of precedence tells us, that relational operators come before the AND operator. Therefore, the expression can be reduced to `1 && 1`, which finally results in `1`.

3. `( (2 > 3 || 1 < 2) && (3 > 2) )`

Here the importance of parens comes into play.

The first part of the expression `(2 > 3 || 1 < 2)` can be reduced to `( 0 || 1 )` which evaluates to `1`.

The second part is reduced to `1`.

The expression is thus reduced to `( 1 && 1 )`, which is `1`.

4. `( 2 > 3 || 1 < 2 && 3 > 2 )`

This is same as the previous one, without the brackets. Might look obvious at first glance, but let's walk through it.

According to the order of evaluation, we first evaluate all the relations. The expression is reduced to `( 0 || 0 && 1 )`.

Next, we evaluate the AND operator, `&&`. The part where the AND operator is used is `0 && 1`, which we know to be `0`.

The expression is reduced to `0 || 0` which is `0`. *See how brackets can change the meaning of an expression?*

5. `( 7 >= 25 || 5 + 5 > 9 && 2 == 3 )`

Let's reduce it like we've done the earlier ones. This time, we have an addition operator which takes precendence and needs to be reduced first. The first step in reduction is therefore `( 7 >= 25 || 10 > 9 && 2 == 3 )`.

Relational comes next. The expression is now reduced to `( 0 || 1 && 2 == 3 )`.

The next step is to reduce equality. The expression becomes `( 0 || 1 && 0 )`.

We now evaluate AND to reduce the expression to `( 0 || 0 )`, which finally results in `0`.