Byte Club's Computer Science with C++

Aryan Mediratta

Contents

Introduction

Hello friend. Welcome to your amazing journey to this new world of Computer Science. This is my attempt to simplify Computer Science to a level at which the information is both decently accurate and easy to understand. I hope you will enjoy learning and find some application of this knowledge in the real life. Computer Science is no less than an adventure to a land you've not been to before, and therefore, I want you to feel the excitement as we begin to take a look at what awaits us.

Another thing I would like to mention is that my goal here is not being highly accurate. Instead, my goal here, is to be easy to understand. When you teach someone math for the first time, you don't begin with Calculus. You start with counting and arithmetic with simple whole numbers. When someone new to mathematics tries to subtract a bigger number from a smaller one, we initially tell them it's not possible (i.e. it makes no sense) and later explain that negative numbers exist.

This is my goal here - Removing all the clutter and going through the foundational building blocks of computer science.

Also, I've attempted to write this book in first person, since I have often noticed that when authors write books in third person, the readers tend to feel a sense of disconnection from both the author and the subject. And that is where I feel most textbooks have gone wrong. This is thus my experiment to try filling this gap between learners and the book.

With that said, I wish you luck for your visit to this new world.

May the code be with you!

Chapter 1

An Introduction to C++

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

Bjarne Stroustrup
The Creator of C++

1.1 Why C++?

This is the very first chapter of the book and I'm talking directly about C++. Is that weird? Many people would have told you that C++ is difficult and that Python is the language to start with. I totally disagree with that opinion. You see, the thing is that Python is so overly simple that if we use it to learn computer science, it won't teach us a lot. You'll realize it later that Python has what I like to call "Magic". And while there's no harm in seeing magic happen, the real fun is in doing that magic yourself. C++ doesn't have much "magic" built-in. This allows us to **do the "magic" ourselves** And trust me, it's going to be awesome.

1.2 Some History

History is boring, isn't it? Well, yes, if you get stuck with the dates and the facts. However, in this case, I'll remove all the boring parts of it. All we care about, is the story of C++. Let's begin.

1.2.1 Assembly

You must have learnt in your previous classes that the computer understands only one language. No prizes for guessing - **Binary**. It's either a zero or a one, i.e. on or off.
However, that's not the complete story. A computer understands binary, but that binary code is actually just a way of writing a language called **Assembly**, also known as **Machine Code**.
But Assembly is not an easy thing to learn. In order to solve this problem, the B Programming language was created.

1.2.2 The B Programming Language

To tackle this problem, Ken Thompson and Dennis Ritchie at Bell Labs created a programming language called **B**. It worked at a higher level of abstraction than Assembly and helped people program using expressions with English words. All code written in this language had to ultimately be compiled (converted) to Assembly in order to be executed. They named it B, considering it a sequel to the BCPL (Basic Combined Programming Language), which was also a very low level language similar to Assembly.

1.2.3 The C Programming Language

Later, Dennis Ritchie created another new programming language called **C**. B was a typeless programming language. C solved this problem by introducing a type system. It remained and is still considered very influential in the world of programming. After some years, a new way (paradigm) of programming emerged. It was called the Object Oriented Programming System (OOPS). C, being an old language, did not support it, although it could do the low level work much better than the new languages.

1.2.4 The C++ Programming Language

Soon, Bjarne Stroustrup at Bell Labs solved this problem by making yet another language called **C++**. It was basically "C with classes". We'll learn about classes later. They are basically the foundational concept behind OOPS. Note that most things valid in C are also valid in C++. The name C++ itself is a nice joke which you'll learn about later in this book.

1.2.5 The people we talked about

Figure 1.1: Dennis Ritchie

Figure 1.2: Ken Thompson

Figure 1.3: Bjarne Stroustrup

1.3 Setting Things Up

We shall now set up our C++ development environment. We will write C++ in an IDE (Integrated Development Environment) called Visual Studio 2019.

It is a very powerful IDE from Microsoft and will help us in easy debugging. At the time of writing this, Visual Studio 2019 is in the Preview (Testing) phase. However, our interface should look more or less the same. If you are reading this when a newer version of Visual Studio (i.e. above 2019) is available, please prefer the latest version.
If you do not like Microsoft or Visual Studio for some reason, you can feel free to use whatever C++ compiler and distribution you like (Suggestion: if not Visual Studio, try CLion - it's free for students). Just make sure you have a debugger.
Also, I must say this in uppercase: STAY AWAY FROM TURBO C++!

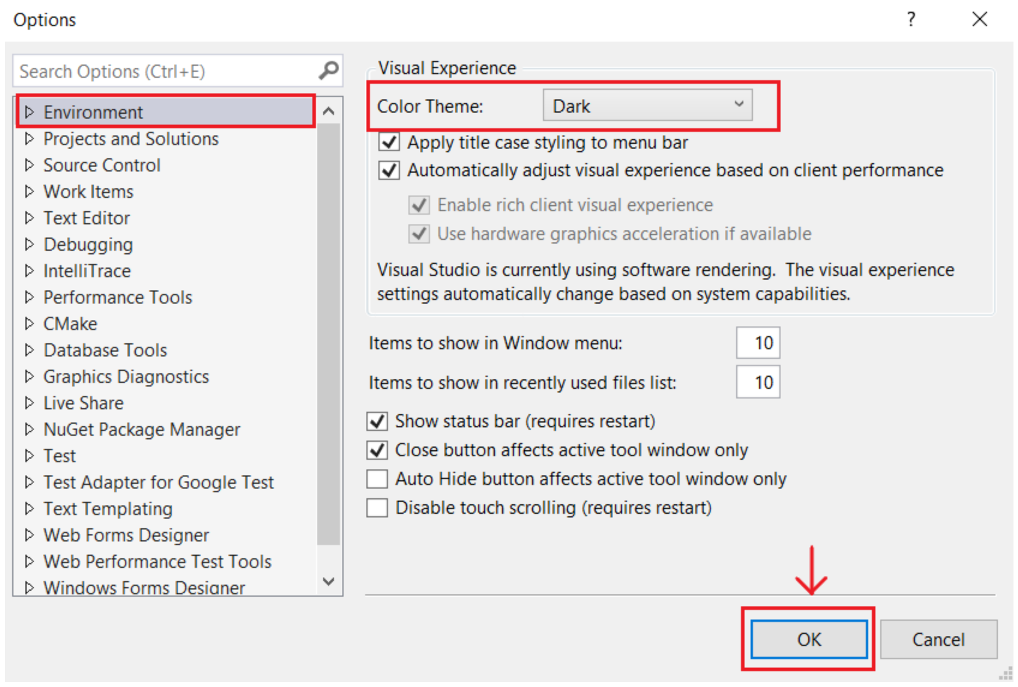
1.3.1 Installing Visual Studio

From a browser, head over to https://visualstudio.microsoft.com/vs/ and download the setup for Visual Studio 2019 Community.
Run the setup and when it asks you for selecting the components, choose "Desktop Development with C++".
Allow it to download and install Visual Studio. It may take quite long, based on your internet connection. However, believe me, it's totally worth the wait :).
If you still need help with the setup, you can scan this QR code and/or visit this YouTube link:

Figure 1.4: (https://www.youtube.com/watch?v=1OsGXuNA5cc)

1.3.2 (Optional) Dark Theme

It is recommended that you switch to the dark theme in Visual studio to prevent eye strain from coding. Most developers prefer the dark theme for this reason - staring at a bright white screen for long can sometimes hurt the eyes. So just for eye-comfort, we prefer the dark theme. Also, it looks cool and makes you appear like a hacker.
Here's how to set up dark theme.
Open a project in Visual Studio (I'll call it VS from now). Open the Tools menu and select "Options".
A setup dialog will open. From the left panel, select "Environment" and now under Visual Experience: Color Theme, select "Dark" from the drop-down. Click OK.



Visual Experience

Color Theme: Dark

- [✓] Apply title case styling to menu bar
- [✓] Automatically adjust visual experience based on client performance
  - [✓] Enable rich client visual experience
  - [✓] Use hardware graphics acceleration if available

Visual Studio is currently using software rendering. The visual experience settings automatically change based on system capabilities.

Items to show in Window menu: 10

Items to show in recently used files list: 10

- [✓] Show status bar (requires restart)
- [✓] Close button affects active tool window only
- [ ] Auto Hide button affects active tool window only
- [ ] Disable touch scrolling (requires restart)

Figure 1.5: Enabling the Dark Theme

1.4 Hello World!

We will now write our first C++ program.
Open Visual Studio. You will be prompted by a window. Select the option that says "Create a new project" (See figure 1.6). Now, select the option

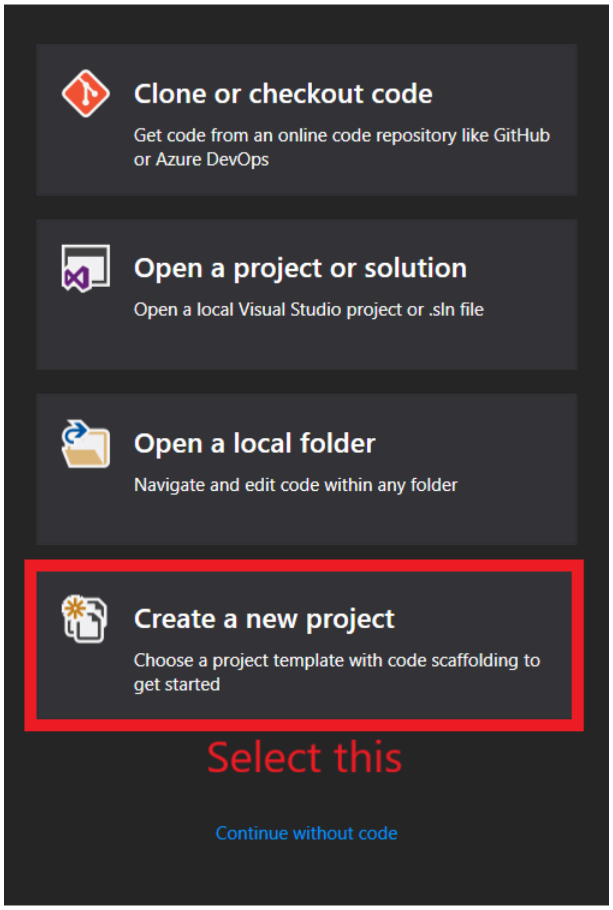
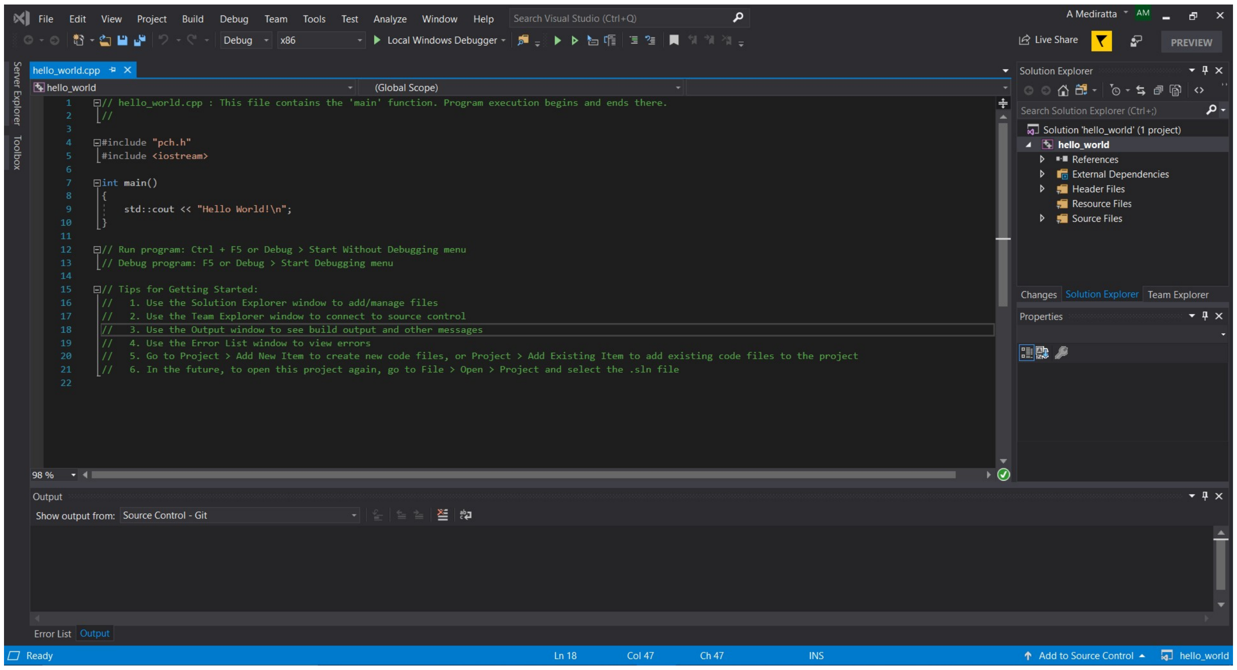

Clone or checkout code
Get code from an online code repository like GitHub or Azure DevOps
Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Select this
Continue without code

Figure 1.6: Select "Create a new project"

that says "Windows Console Application". Click Next. Give your project a name and click on "Create". You should see something like this: (See figure 1.7) Now, delete everything other than the line that says `#include "pch.h"` (or you might see `#include "stdafx.h"`). We will talk about this later. It is not a part of the language itself - it is something specific to VS.
Now, enter the following code after that line:



```
// hello_world.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include "pch.h"
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln file
```

Figure 1.7: New Project in Visual Studio

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!" << endl;
}
```

Now, press CTRL+F5. You will see a console window with the text “Hello World!”. Yeah! We made a computer display “Hello World!” on screen. Let us now try to understand what each part of the code does.
Output:

```
Hello World!
```

1.5 Dissecting Hello World!

The Preprocessor Include Directive

Before I say anything about the preprocessor include directive, please understand that you do not need to worry about the name. It's just a fancy name for something really simple.
`#include` is the preprocessor include directive.
A preprocessor directive is an instruction which are executed **before** your code is compiled.
In this case, we used the include directive to automatically copy and paste all the code that is there in a C++ library (header file) called `iostream`.
All preprocessor directives start with a `#` symbol (You can call it the sharp, the pound, the number symbol, the hash, the hashtag or even the octothorpe; I just call it the hash symbol).

The `iostream` Library

The `iostream` header allows us to take input from and give output to the user. In our case, it provided us with the functionality to use `cout` which allows us to display text on the screen and `endl` which ends the line.

The Standard Namespace

All C++ standard library features are enclosed in a container called the standard namespace or `std`. The `using namespace std;` part of the code allows us to tell the compiler that we are using the standard namespace. If we choose to not use that line in our code, we'd have to refer to all `std` features with `std::`, i.e. `cout` would become `std::cout` and `endl` would become `std::endl`.

The `main()` function

The main function (`int main()`) defines what's called the entry point of the program. Code execution starts and ends with the main funcion.

The Curly Braces

Curly braces {} in C++ define the start and end of something. In the Hello World program, you can see that we used the curly braces to define the start and the end of the main function. Anything that is enclosed within the curly braces is commonly called a “block” of code.

The Semicolon

Semicolons (;) in C++ denote the end of a complete statement. C++ gives us the ability to say a single statement in multiple lines and multiple statements in the same line. So when we are done with a complete statement, we use a semicolon to tell the compiler, “Hey, I’m done describing this instruction. Whatever you see after this semicolon will be related to the next instruction.” You can roughly think about it as being analogous to the ‘full stop’ (.) punctuation mark in English. (And French. And German. And other languages.)

Chapter 2

An Overview of Computer Science

“Computer Science is no more about computers than astronomy is about telescopes.”

Edsger Dijkstra
Inventor of the Shortest Path Algorithm

This quote perfectly describes what most people think Computer Science is and shows where they go wrong. Computer Science is not about learning how to use computers. It is instead about computation - about understanding fundamentally how a computer processes information - and about how we can make it better. It deals with the study of how efficient some algorithms are and how we can increase their efficiency.

2.1 What is a computer anyway?

Before I answer this question, I want you to think about where the word ‘computer’ even comes from. Think about words that look or sound similar to the word ‘computer’.
Perhaps you thought of the verb ‘compute’ and/or the noun ‘computation’. But now, if you look up a dictionary, to compute means to calculate. From that analogy, a computer must be a calculator, right? Here’s what my teacher told me about a computer when I was in the third grade:

Definition 1 *A computer is an electronic device that takes input, processes it, and gives an output.*

While this definition is quite elegant and describes what a computer appears to do at the surface, we can go deeper.

2.1.1 The Turing Machine

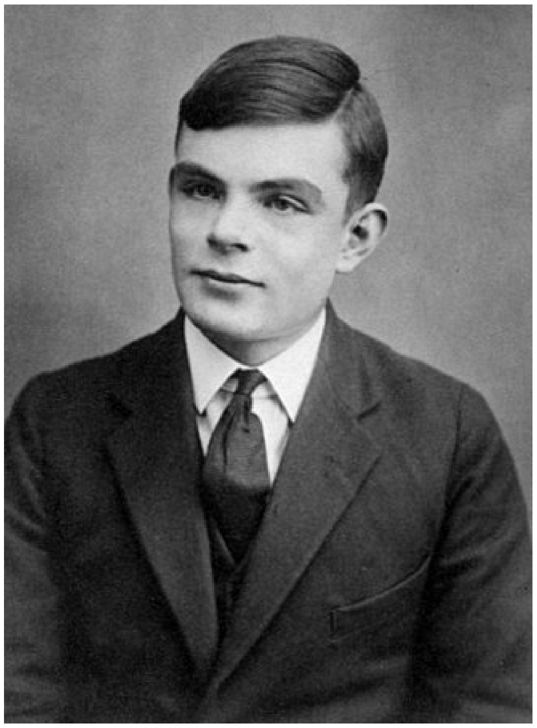


Figure 2.1: Alan Turing

Say “Hello” to Alan Turing! He was a great computer scientist... Oops! He was not just a computer scientist. Wikipedia describes him as an English mathematician, computer scientist, logician, cryptanalyst, philosopher and theoretical biologist (Whoa! That’s way too many professions at once.).
So he is well known for his definition of a concept now known as the Turing Machine. A turing machine can be thought of as a hypothetical (imaginary) machine performing operations on data represented as a long series of zeroes and ones (which he described as a tape with blocks or cells that contained

either a zero, or a one, or nothing).
Now, a turing machine can do the following things based on the instructions given to it:

- Read what is stored at a particular cell. [Note that it can only be at one cell at a time.]
- Move left or right towards any given cell.
- Erase data from a cell.
- Write 0 or 1 at a given cell.

The initial state of the data is the **input** and what is left of the data after the Turing Machine has followed the instructions, is the **output**. Now, while it might look most tasks a computer does can more-or-less be understood using the Turing Machine.
Scan this QR code to learn more about Turing Machines:
You will now see how the definition of the Turing Machine simply coincides

Figure 2.2: (https://www.youtube.com/watch?v=dNRDvLACg5Q)

with what we already know about. We will now revise some terms we have already learnt.

2.1.2 Things we already know

Definition 2 *The information a computer takes from the user for processing is called input.*

Definition 3 *The information a computer gives back to the user after processing is called output.*

The CPU has 2 main units:

- ALU (Arithmetic and Logic Unit): it does all the math and makes the decisions.
- CU (Control Unit): it controls the input and output devices.

2.2 Types of Knowledge

In computer science, we divide knowledge into categories and primarily study a specific category of knowledge.
Knowledge exists in two types - Imperative and Declarative

Definition 4 *Imperative knowledge is the type of knowledge that describes a procedure to do something - usually as a set of instructions.*

Definition 5 *Declarative knowledge is the type of knowledge that states facts about something.*

2.3 Things to Keep in Mind

- Computers are deterministic machines.
- The output of a program is mostly predictable, unless it involves any randomization.
- They do exactly what they're told.
- If they don't do what you expected, it means you didn't tell them the right thing to do.
- In computer science, we study instructions given to the computer as algorithms, often represented as a flowchart or mostly using what's called pseudocode.
- Computers are dumb. However, they are very good at doing calculations and following your instructions quickly.
- A computer can do two types of calculations:
  - The ones already defined in the language.
  - The ones you define yourself.

Chapter 3

Talking About Algorithms

"An algorithm must be seen to be seen to be believed."
Donald Knuth
The creator of the TeX computer typesetting system

3.1 Motivation

We often have to tell the computer what to do. But what if we want to tell the computer how to do what needs to be done? How can we tell a computer the procedure to do something? Obviously, before we can tell the computer how something must be done, we should know that procedure in the first place.

3.2 What is an Algorithm?

You can think of an algorithm as a well-defined computational procedure that takes some value(s) as inputs and returns some value(s) as output. It is like a mathematical device used to solve a computational problem. Algorithms can be represented in many ways, such as a computer program, a flowchart or even structured English (or any other language).
A highly effective analogy to understand an algorithm is a **recipe**. Just like a recipe, an algorithm has a series of well-defined steps that help in solving a problem.

YOU CAN'T TELL A COMPUTER HOW TO DO STUFF
IF YOU DON'T KNOW HOW IT'S DONE IN THE FIRST PLACE
imgflip.com

Here's a simple example of an algorithm to call a friend using a telephone.
Input: Your friend's telephone number.
Steps:

```
Pick up the receiver.
Dial the number received as input on the keypad.
Wait until you hear it ringing. Continue when you no
    longer hear the ringing sound.
If you hear your friend, go to step 6.
If you do not hear your friend, end the call. Wait 5
    minutes. Go to step 1.
Talk to your friend.
End the call.
Halt.
```

3.3 Making a computer follow an algorithm

Now that we know what algorithms are, let me try to show you the various forms an algorithm can take when you try to tell a computer the steps it needs to follow.

3.3.1 Language Processors

We discussed in Chapter 1 that a computer understands only machine code. This machine code is specific to every processor. Therefore, we have a

language called Assembly that gets translated to system-specific machine code. This piece of software that translates assembly into specific machine code. We call it the **Assembler**.
Now, Assembly is not an easy thing to write. So we mostly use high level languages to program a computer. All high level languages are either compiled (compiler-based) or interpreted (interpreter-based). C++ is an example of a compiled language and Python is an example of an interpreted language.
But what exactly is the difference?
Here's how an interpreter works.
It reads the code you wrote **line-by-line** and translates each line into machine code. So it starts by executing the instructions of the first line. When that is done, it translates and executes the code of the second line, and so on...
Now, a compiler is what I consider smarter than an interpreter for this reason:
A compiler takes **all your code at once**, and translates the whole thing into machine code. Then you can manually (or automatically) execute that compiled code whenever you like.
Another advantage of a compiler over an interpreter is that, if there is an error in the code you wrote, the compiler will halt and display where each error is. However, an interpreter won't even find an error on line 485 unless it has executed everything before line 485.
Here's a quick summary of what we discussed:

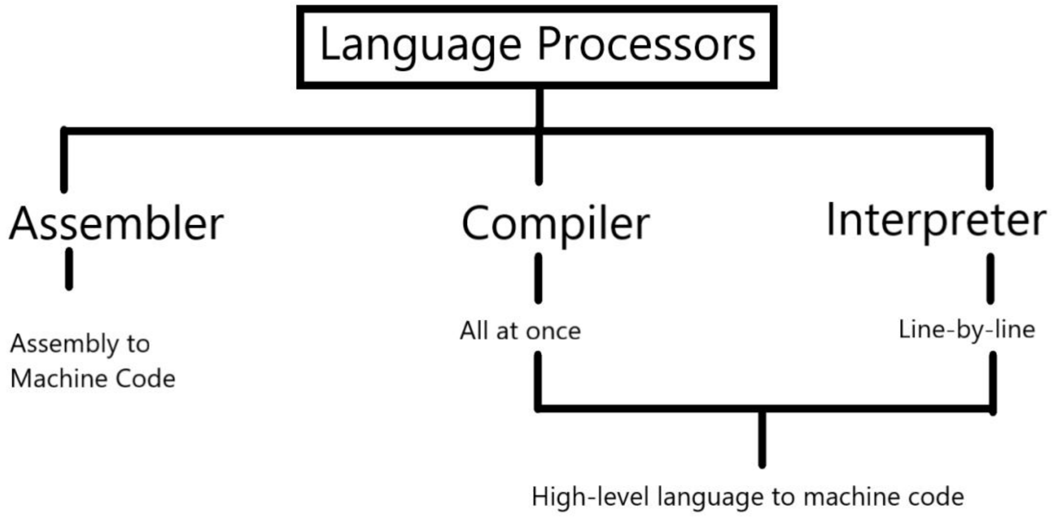


Figure 3.1: Language Processors - Summary

Chapter 4

Boolean Algebra

"One should never try to prove anything that is not almost obvious."
Alexander Grothendieck
French Mathematician, 1966 Fields Medallist

Boolean algebra is a concept of mathematics that helps us deal with statements about truth and allows us to formally talk about what is true and what is false.
Don't worry, I assure you that this is way easier than the mathematics you have studied until now. In fact, most of what we discuss here will seem to be common sense to you. And it is indeed common sense. It's just that we need a mathematical framework to talk about common sense.
In Computer Science, this concept is mostly referred to as "Logic".
Yeah, we have a real defined meaning of the word "Logic". Logic is simply a way of dealing with things that are either true or false.

4.1 Some Background

Boolean algebra is known to have been invented by George Boole (hence, the name "Boolean").
But wait, did you notice how things in Boolean Algebra seem to be of two states (true and false), just like how binary is of two states (1 and 0)?
Even if you didn't, do these ideas look connected now?
Let me tell you that they are indeed well connected. Fun fact: One of

the reasons why we made computers use the binary system is that Boolean Algebra existed much before computers did, and this helped us connect true to 1 and false to 0.
I hope you can now appreciate why we must study this strange branch of mathematics as a part of Computer Science.

4.2 Variables

A boolean variable can only have two values, a zero or a one. I will denote boolean variables as lower-case English letters, such as a, b, c, et cetera.

4.3 A simple warm-up

Before I explain how things work in Boolean Algebra, I want to show you how our intuition perfectly matches with what we're going to see.
As a warm-up exercise, state whether the following statements are true or false.

1. $1 + 1 = 2$
2. $90 \leq 100$
3. $1 + 5 > 9$ OR $2 + 2 = 4$
4. $1 \times 0 = 100$
5. In a right triangle, the square of the hypotenuse is NOT equal to the sum of the squares of the other two sides.
6. Either you are wearing a red shirt OR you are NOT wearing a red shirt.
7. If you toss a coin AND it lands on its circular face (not on its edge), then you will either get a Heads or a Tails.

If you did this correctly like most people, you'll say that statements 1, 2, 3, 6 and 7 are true and the rest are false.

4.4 Gates

In Computer Science, a boolean operation is called a gate.
A gate can be looked at as something that takes one or two booleans as input and gives out a single boolean as an output.
If you study Boolean logic in Mathematics, you will find some complex notation. I will not use that notation here, so that you don't find yourself lost in the cumbersome notation and just get the idea of how things work. I hope that helps.

4.4.1 The AND gate

Don't worry, AND is not an abbreviation. It is literally the English Word "and" written in ALL CAPS.
Boolean Algebra is best studied by making tables called **Truth Tables**.
Let us look at the truth table for the AND gate. Remember that 0 is false and 1 is true.

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

This describes the behaviour of the AND gate. The AND of two booleans is true if and only if both of them are true, i.e. a AND $b = 1 \Leftrightarrow a = 1, b = 1$.
If you think about it, this idea makes perfect sense. The whole statement "I live in Delhi AND I know C++" is true if and only if both the individual statements "I live in Delhi" and "I know C++" are true. If any of these two statements is false, then the whole statement is false.

4.4.2 The OR Gate

Again, this is just the English word "or", written in uppercase, not an abbreviation.
Let's look at the truth table of the OR gate.

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

From the table, we can see that the OR of two booleans is true if at least one of the two booleans is true. Even if both the booleans are true, the OR of the two booleans is true.
In more precise terms, this is the Inclusive OR Gate.

4.4.3 The XOR Gate

XOR stands for eXclusive OR.
Unlike the inclusive OR, the XOR gate returns false if both the statements are true.

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

It returns true if out of the two statements, strictly one of them is true. If both are true, the overall XOR statement is false.

4.4.4 The NOT gate

Unlike the other gates, the NOT gate takes only one boolean as input and returns the negative of that input, i.e. NOT true = false and NOT false = true.

a	NOT a
0	1
1	0

4.5 Boolean Identities

The proofs of these identities are trivial. You can prove these by making a truth table for the LHS and the RHS. You do not need to remember any of this, but make sure you understand what they mean, or at least verify if they look obvious (spoiler alert: they will).
WARNING: DO NOT TRY TO REMEMBER THEM. Make them a part of your life.

$$a \text{ OR } b = b \text{ OR } a \tag{4.1}$$
$$a \text{ AND } b = b \text{ AND } a \tag{4.2}$$
$$a \text{ XOR } b = b \text{ XOR } a \tag{4.3}$$
$$a \text{ OR } (b \text{ OR } c) = (a \text{ OR } b) \text{ OR } c \tag{4.4}$$
$$a \text{ AND } (b \text{ AND } c) = (a \text{ AND } b) \text{ AND } c \tag{4.5}$$
$$a \text{ AND } (b \text{ OR } c) = (a \text{ AND } b) \text{ OR } (a \text{ AND } c) \tag{4.6}$$
$$a \text{ OR } 0 = a \tag{4.7}$$
$$a \text{ OR } 1 = 1 \tag{4.8}$$
$$a \text{ AND } 0 = 0 \tag{4.9}$$
$$a \text{ AND } 1 = a \tag{4.10}$$
$$a \text{ AND } (a \text{ OR } b) = a \tag{4.11}$$
$$a \text{ OR } (a \text{ AND } b) = a \tag{4.12}$$
$$(\text{ NOT } a) \text{ AND } (\text{ NOT } B) = \text{ NOT } (A \text{ OR } B) \tag{4.13}$$
$$(\text{ NOT } a) \text{ OR } (\text{ NOT } B) = \text{ NOT } (A \text{ AND } B) \tag{4.14}$$
$$a \text{ AND } (\text{ NOT } a) = 0 \tag{4.15}$$
$$a \text{ OR } (\text{ NOT } a) = 1 \tag{4.16}$$

4.6 Boolean Algebra and Set Theory

You will realise in class 11 mathematics that most of what we just learnt is basically applied in another completely different field in mathematics called Set Theory.
The idea of OR translates to the union of two sets. The idea of AND translates to the intersection of two sets and the idea of NOT translates to the complement of a set.
The identities we just learnt will have a direct application in both programming

and set theory. In fact, most of these identities (other than the ones involving 0 and 1) are basically the same as those they teach in Set Theory.
Each of these identities has a name, but I don't want to confuse you with that :).

Chapter 5

C++ From Scratch

"There are only two kinds of languages: the ones people complain about and the ones nobody uses."
Bjarne Stroustrup
The Creator of C++

5.1 The language

C++ is a compiled language. This means that there exists a piece of software called the compiler that takes your source code and "compiles" it into machine-readable code in an executable format supported by the operating system. For Windows, this is the .exe executable format. For *nix based systems, this is mostly the .out format.
The standard libraries are huge repositories of code written in perfectly normal C++ and provide us with various functionalities. However, there are other entities in C++ which are built into the language itself.

5.2 The Minimal Program

The minimal C++ program looks like this:

```
int main() {}
```

This is the essential bare-minimum for a program to be valid C++. What does this code do? Nothing. It literally just declares a **function** called "main" and does not define anything inside the function.
The curly braces {} denote the start and end of a code block. In this case, the code block is used to enclose the contents of the `main()` function.

5.3 Console Output

In our first C++ program, we made the computer display "Hello World!" on the screen. For a recap, this was the program:

```cpp
#include <iostream>
int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

In the beginning, we included a header file called `iostream` which enables us to display output to and take input from the user.
Now, here's what I want you to try. You know that we made the computer display "Hello World!". It must be visible to you that we could otherwise make it display anything we like. So, go ahead and try to replace that text inside the quotes with something else of your choice and see what happens when you compile and run it. If you successfully changed the text, it would have displayed the new text you changed.
Great job! You can now make the computer display whatever you like.

Remark 1 *In fact, the name `iostream` literally speaks for itself. I/O is the abbreviated form of Input/Output.*

Here's another experiment I want you to try.
Remove the quotes surrounding the text and now try to compile the program. You will now see an error. (depending on which distribution of C++ you use). That's sad :(
But now, try replace that text with 2+2, i.e.

```cpp
#include <iostream>
int main()
{
    std::cout << 2+2 << std::endl;
    return 0;
}
```

You'll see that this program works, even without the quotes and in fact, this time, it displayed 4.
`std::cout` is a **keyword** in the `iostream` library that is used to display output to the console and `std::endl` is used to end the line.

5.4 Comments

We often have to mark certain parts of our code so that we remember what that does or so that other people can understand our code easily. This is done through comments.
Any part of the code that is commented is not read by the C++ compiler and is ignored. It only helps humans understand the code better.
There are two types of comments allowed in C++: Single-line comments and Multi-line comments.

5.4.1 Single-line comments

Single-line comments, as the name suggests, span upto the current line. Single-line comments made with two forward slashes (//).
For example:

```cpp
// This is a single-line comment
// This is another single-line comment
```

Note that single-line comments can start from any point in the line, and extend for that whole line.
Therefore, if a single line comment can start right after a piece of code and that code will still be executed.
Example:

```cpp
std::cout << "ABC" << std::endl; // comment
```

5.4.2 Multi-line comments

Multi-line comments start with a /* and end with a */.
Example:

```
/* This is
a multi-line
comment */
```

5.5 Variables

When you read this word “Variable”, it might remind you of algebra. However, the notion of a variable is similar, yet quite different in computer science compared to what you’ve learnt in algebra.

5.5.1 A Useful Analogy

With time, you will realise for yourself what a variable actually is. However, since you are probably new to programming in general, here’s a useful analogy that works. Whenever you’re stuck, this analogy will help you.
Whenever I say, “Variable”, imagine a **bucket** that can store a particular type of information. We can read the information stored in a variable (bucket) and change it. Later, we’ll refer to it as space in the memory but for now, just think of a bucket.
This idea is very important - We can change the data stored in a variable by assigning it a new value.

5.5.2 Declaring Variables

Now that you know what I mean by variables, we can now start playing with variables.
Here’s how we declare a variable in C++.
We first mention the keyword corresponding to the datatype, i.e. the type of information we want to store in the bucket and then the name of the variable.
These are some of the datatypes C++ supported by default:

Keyword	Datatype
bool	Boolean (true or false)
int	Integer (Ex. 2, 1000, -282)
char	Character (Ex. 'a', '@', 'N')
float	Floating-point decimal (Ex. 2.292)
double	Double-precision floating point decimal (Ex. 39.329034)

Example variable declaration:

```cpp
int myNumber;
```

The assignment operator (=) is used to assign a value to a variable.
Example:

```cpp
bool isHappy; // Creates a boolean variable
isHappy = true; // Assigns value true to isHappy
```

The two lines of code above can combined into a single statement.

```cpp
bool isHappy = true;
```

Like in the above line, when we create a variable and assign it a value within its declaration, it's called initialization. It's considered a good practice to always initialize your variables with a value. Another way to initialize a variable is the following:

```cpp
int myNumber {1};
```

Note that although you can initialize a variable like this, you cannot assign a value to variable using curly braces. For assignment, the equal sign must be used.
Also, by the modern C++ Core Guidelines, it is considered as a good practice to use curly braces for initialization (i.e. the last method).

5.6 Constants and the const Keyword

A constant is just like a variable. The only difference is that you can assign a value to a constant only once. Constants are declared just like variables; it's just that they have the `const` keyword right before the declaration.
The `const` keyword is like telling the compiler: "Hey! I promise I won't try to change the value of this variable later." And if you break the promise, the compiler will give you an error. It's considered as a good practice to

use the `const` keyword whenever you're sure you won't have to change the value held by that "bucket". Note that a constant declaration must have an initialization. You CANNOT leave a constant uninitialized.
Example:

```
const int a = 1; //You cannot assign a to anything else now. You
 can only use the value of a.
```

5.7 Operators and Operands

You know what an operator is in mathematics. We have four main arithmetic operators in math for addition(+), subtraction(−), multiplication(×) and division(÷). Similarly, we have operators in C++.

Definition 6 *An operator is a symbol that tells the computer to perform some specific operation.*

Definition 7 *Operands are the values that the operation performed by an operator takes.*

Unlike most other languages, C++ allows us to have custom operators. However, we won't discuss custom operators here.
C++ has many operators built-in.
Let's look at some of them.

5.7.1 The assignment operator

The equal sign (=) is the assignment operator in C++, as discussed before. It assigns values to variables. However, it must not be taken like the equality symbol in mathematics. What you are going to see now is a valid C++ program.

```
#include <iostream>
int main()
{
    int a {1};
    a = 6; // Yes, this is valid.
    std::cout << a << std::endl;
    return 0;
}
```

What do you think will be the output of this program? The variable a had been assigned a value before being assigned a value of 6. Now if you see it like mathematics, it doesn't make sense. We can't say $a = 1$ and $a = 6$ at the same time.
This is where the bucket analogy will help. Consider a bucket that had the number 1 in it. Then, it was emptied and 6 was put in it.
Therefore, the output of this program will be 6.

5.7.2 The arithmetic operators

The addition operator

As the name suggests, the addition operator can add two variables or literal numbers.
The symbol for addition is the 'plus' sign (+).

Remark 2 *The plus sign (+) does not only do the work of addition. It has multiple uses, as we will see later.*

Look at this example:

```cpp
#include <iostream>
int main()
{
    int a {6};
    int b {5};
    std::cout << a + b << std::endl; // This will
       output 11
    return 0;
}
```

If you compile and run this, you'll get the output as 11. The reason should be obvious. We know that when we add 6 and 5, we will get 11.
Now, think about this. What will this piece of code do?

```cpp
#include <iostream>
int main()
{
    int a {6};
    a = a + 2;
    std::cout << a << std::endl;
    return 0;
}
```

We made a new variable (bucket) and gave it the value 6. Then, we said `a = a + 2;`. If this confuses you, don't worry. Just remember that this is not an algebraic equation. It is an **instruction** to the computer. What it does is that it sets the value of the variable `a` to a new value which is exactly two more than its initial value.
Therefore, the output of the above program will be 8, since $6 + 2 = 8$.
We will soon find that there is another operator to do exactly that, and requires us to type less. However, if I had just shown you what that operator did, you'd never know anything like `a = a + 2;` is possible. That's the fun of learning. First, you learn it the hard way; and then, once you've understood how things work, you learn it the easy way.

The Subtraction operator

Again, you probably know what it does, but let me still say it.
The subtraction operator finds the difference between the first and second values it is provided with.
The hyphen symbol (or the minus sign), $(-)$ is the subtraction operator.
Look at this example:

```
#include <iostream>
int main()
{
    int a {6};
    int b {5};
    std::cout << a - b << std::endl; // This will
       output 1
    return 0;
}
```

Again, we know that $6 - 5 = 1$, therefore, the output of this program will be 1. Note that if a were smaller than b, you would've got a negative number (obviously).

The Multiplication operator

The multiplication operator multiplies the two types of numbers it receives.
The asterisk symbol ($*$) is the multiplication operator in C++.
This example will explain that:

```
#include <iostream>
int main()
{
    int a {6};
    int b {5};
    std::cout << a * b << std::endl; // This will
       output 30
    return 0;
}
```

Clearly, $6 \times 5 = 30$, therefore, the program will output 30.

The Division operator

The division operator divides the two types of numbers it receives.
The forward slash symbol (/) is the division operator in C++.
However, there's a small catch. When the division operator is applied on two integers, it returns an integer. This means, you will only get the **quotient** of division.
Though it will return a decimal answer if it is supplied with two `double`s or `float`s. This example will explain that:

```
#include <iostream>
int main()
{
    int a {6};
    int b {5};
    std::cout << a / b << std::endl; // This will
       output 30
    return 0;
}
```

Clearly, $\frac{6}{5}$ gives quotient 1, therefore, the program will output 1.
But now look at this:

```cpp
#include <iostream>
int main()
{
    double a = 5.0;
    double b = 2.0;
    std::cout << a / b << std::endl; // This will
       output 2.5
    return 0;
}
```

The output of this program will be 2.5. This is because we are dividing two floating point decimal numbers.

The Modulo Operator

The modulo operator finds the **remainder** when the two numbers received by it are divided.
The symbol for this operator is the per cent sign (%). Note that this operator is only defined for integers.

```cpp
#include <iostream>
int main()
{
    int a {8};
    int b {3};
    std::cout << a % b << std::endl; // This will
       output 2
    return 0;
}
```

The output of this program will be two because when you divide 8 by 3, the remainder you get is 2, because $8 = (3 \times 2) + 2$.

5.7.3 Boolean Operators

Remember we studied boolean algebra (see Chapter 2)? The logic "gates" we studied there are built-in operations in C++.

The OR operator

The OR operator takes the OR of two boolean values.
The symbol for it is two vertical pipe symbols (||).
Look at this program:

```
#include <iostream>
int main()
{
    bool a {true};
    bool b {false};
    std::cout << a || b << std::endl;
    return 0;
}
```

You will see that this program outputs 1, which is the same as the boolean 'true'.

Other Boolean Operators

By now, you would've got an idea of how operators work. So let me just show you a quick table of all C++ boolean operators, so that you can try to use them.

Operation	Operator
AND	& &
NOT	!

And well, that's it! The XOR gate is just a combination of AND, OR and NOT.
Basically, a XOR b = [a AND (NOT b)] OR [b AND (NOT a)]

5.7.4 Comparison Operators

Here's a table of the comparison operators in C++. You'll find them quite simple. They return a boolean value (`true` or `false`), based on whether the whole statement is `true` or `false`.

Operator on x and y	Comparison
x == y	equal
x != y	not equal to
x < y	less than
x > y	greater than
x <= y	less than or equal to
x >= y	greater than or equal to

5.7.5 Other operators

Operator on x	Equivalent to
$++x$	$x = x + 1$
$--x$	$x = x - 1$
$x+ = 2$	$x = x + 2$
$x- = 3$	$x = x - 3$
$x/ = 4$	$x = x/4$
$x\% = 4$	$x = x\%4$

For information on how these work, please scan this QR code and watch the video:

Figure 5.1: (https://www.youtube.com/watch?v=T0kEDZ-tuNw)

Hopefully, after watching the video, you can now understand the meaning of the name "C++". It was called C++ because it's more than just C. It's like an **increment** to what C was. Although C++ and C are very similar, C++ has some nice advanced features built-in. Also note that C++ is backwards compatible with C. This means that most things that work in C also work in C++.

5.7.6 End Note on Operators

These are not the only operators in C++. We will talk about more operators later. In fact, C++ is so flexible that it even allows custom operators.
Also, you can always use multiple operators in the same instruction. When you do that, you can use parentheses, i.e. round brackets () to specify which operation is to be done first. For example:

```cpp
#include <iostream>
int main()
{
    std::cout << ((2 + 2) - 1) << std::endl;
    return 0;
}
```

This program will output 3 since 2 + 2 is 4. Subtract one, and that's 3. (Quick Math!)
So the output will be 3.

5.8 Console Input

You can take the value of variables as input using `std::cin`.
Here's how you can do it.

```cpp
#include <iostream>
int main()
{
    int a {};
    std::cout << "Enter a number: " << std::endl;
    std::cin >> a; // Takes a as input
    std::cout << "You entered " << a << std::endl;
       // Outputs the value held by a
    return 0;
}
```

5.9 The `if` Conditional Statement

5.9.1 The Simple `if` Statement

The `if` statement takes a boolean expression and a block of code to be executed if the boolean expression evaluates to true.
I know that doesn't sound simple, but it's really not that hard. Let me just show you an example.

```
#include <iostream>
int main()
{
    int a {5};
    if(a > 3)
    {
        std::cout << "a is greater than 3." << std::
           endl;
    }
}
```

In the above example code, look at what we're doing.
We initialized a variable `a`. Then in the `if` statement, we checked if the value of `a` is greater than 3.
The if statement told the computer check if a is greater than 3 and to execute the instructions inside the `if` block if the condition is true.
So the output will be:

```
a is greater than 3.
```

5.9.2 Extending the `if` statement

Let us quickly look at how multiple conditions are evaluated.
The `else if` statement contains the condition that is checked if the previous if condition was not true. Here's an example:

```
#include <iostream>
int main()
{
    int a {-8};
    if(a > 0)
    {
        std::cout << "Positive" << std::endl;
    }
    else if(a < 0)
    {
        std::cout << "Negative" << std::endl;
    }
}
```

The output of the above program will be `Negative`. It first checked if `a` was greater than zero. But that's false. So it moved on to check the next condition provided by the `else if`.
You can also use the `else` block in this way:

```
#include <iostream>
int main()
{
    int a {-8};
    if(a > 0)
    {
        std::cout << "Positive" << std::endl;
    }
    else if(a < 0)
    {
        std::cout << "Negative" << std::endl;
    }
    else
    {
        std::cout << "Zero" << std::endl;
    }
}
```

The output of the above program will be `Zero`. The else block is executed when none of the above statements is true.
Note that you can also have an `else` block directly after an `if` block without the `else if`.

5.10 Iteration or Looping

What do you think iteration means? Perhaps you'd think it's related to doing something many times. And you'd be right. It is exactly that. Now the question is, if I want a computer to do something five times, why should I learn about iteration/looping when I can simply copy and paste that piece of code five times?
Here's the idea. That may work if you know at the time of programming, how many times something has to be repeated. But what if that number were to be taken into a variable as input? That's where the concept of loops comes in.
(Warning: NEVER attempt to repeat instructions by simply copying and pasting them in place. Doing that without using a loop is a HORRIBLE way of programming called WET code)
Loops help us overcome the problem of WET code (i.e. Write Everything Twice) and help make our code DRY (i.e. Don't Repeat Yourself).
There are many types of loops in C++.

5.10.1 The `while` Loop

The `while` loop takes a condition enclosed within a pair of parentheses () and keeps repeating the piece of code inside its block as long as the condition of the loop holds true.
For example, if you had to display all the natural numbers from 1 to 4, you could do it this way:

```cpp
#include <iostream>
int main()
{
    int a {1};
    while(a <= 4) // Repeat while a is less than or
        equal to 4
    {
        std::cout << a << std::endl;
        a++; // Increment a by 1
    }
    return 0;
}
```

The idea is that a `while` loop, before every iteration, checks whether the statement in the parentheses is true. Then it executes the code inside the curly brackets if the statement is true and stops looping if the statement is false.
Therefore, the above program will have the following output:

```
1
2
3
4
```

5.10.2 The `for` Loop

The above example can be simplified into a different kind of loop called the `for` loop. It takes three statements in the parentheses, an initialization, a condition, and an increment/decrement. Here's how you'd print numbers from 1 to 4 using a `for` loop.

```cpp
#include <iostream>
int main()
{
    for(int a {1}; a <= 4; a++)
    {
        std::cout << a << std::endl;
    }
    return 0;
}
```

This program will have the same output as the previous one.

Chapter 6

Efficiency of Algorithms

“The purpose of computation is insight, not numbers.”
Richard Hamming

What is efficiency? Intuitively, it’s just a measure of how efficient something is. If a vehicle is fuel-efficient, it can travel longer distances with the same amount of fuel, compared to a vehicle which is not fuel-efficient.
Similarly, algorithms can also be efficient or inefficient based on how many calculations or computations they have to make in order to solve a problem. An algorithm that solves the problem in fewer calculations is considered time-efficient.
Also, an algorithm that solves the problem by using a lower amount of space in memory is space-efficient.
In this chapter, we will talk about the time-efficiency of algorithms. Actually, this time, you will teach yourself efficiency. Have a look at this link:
https://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/
Hopefully, that explains the whole topic.

Chapter 7

Further Resources

TBA (To be added)