Programming Fundamentals in C Part I

1. What is Programming?

A computer program is the design and building of algorithmic processes in order to achieve a specific goal. Their structure is often reliant on 2 main foundations:

- **Data** – All programs fundamentally work with data. Without data, programs are empty husks that do not have any meaning. In programming, we store data in what we call **variables**.
- **Method / Process** – In what way are we manipulating our data to achieve our goal? In programming, we call these **functions**.

2. Introduction to C

C is a low level language that is useful for the learning of programming because of the following:

- Basic code structure
- Lesser abstractions such that fundamentals are easily taught

Advantages:

1. Code runs faster.
2. Building block for many other programming languages (coined the mother of all languages).
3. Excellent medium to teach fundamentals of programming and ultimately, computer science.
4. Once you learn C, you will be versatile in learning almost every other language out there.

3. Binary

In computers, every piece of data is read in binary, meaning that it is represented solely in 0s and 1s.

Bit

In computer architecture, information is processed through electricity, so how are signals being created? In computers, think of components as switches, it is either "on", or it is "off". To abstract this, we use what is known as a **bit**, which takes two values.

0 ("off") or 1 ("on")

Arithmetic with bits

DECIMAL

1	**2**	**8**
10^2	10^1	10^0
1*100	**2***10	**8***1

= 128

BINARY

1	**1**	**0**	**1**
2^3	2^2	2^1	2^0
1*8	**1***4	**0***2	**1***1

= 13

The idea by which we represent and count numbers are the same for both decimal and binary, except that binary numbers only have 1s and 0s while decimal numbers have numbers 0 to 9!

Binary: Only contains **2** numbers and carryovers represent powers of 2.

Decimal: Contains **10** numbers and carryovers represent powers of 10.

Exercise:

Convert the following binary numbers into its decimal equivalent:

1. **100101:** _____
2. **100111:** _____
3. **1101:** _____

Note: A byte is simply 8 bits. (0000 0000 – 1111 1111) **or** (0 – 255)

Byte

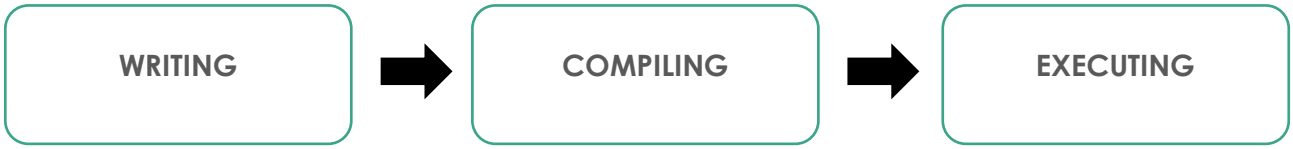
A byte is simply 8 bits, meaning the number 255 (**decimal**) or 1111 1111 (**binary**).

Often times, in computers, we refer to data in terms of bytes.

1	**1**	**1**	**1**	**1**	**1**	**1**	**1**
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1*128	**1***64	**1***32	**1***16	**1***8	**1***4	**1***2	**1***1

= 255

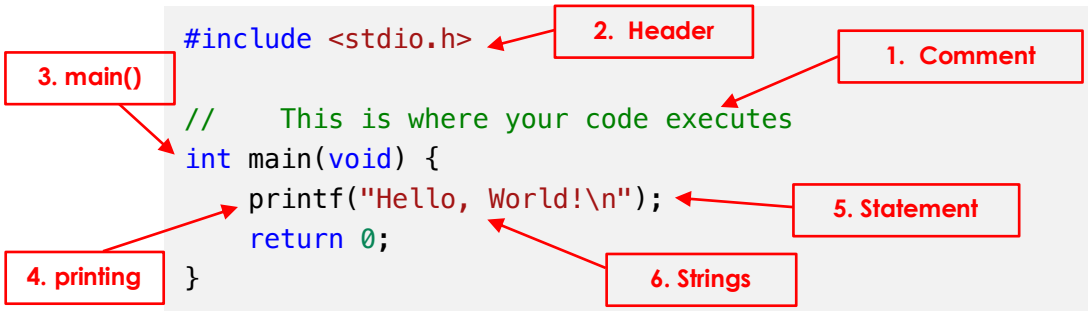
4. Running a C Program



Creating a C file

To write a program in C, we must first create a file with the **.c** extension. Inside your directory in VS Code, create a new file called **hello.c**.

Writing your first program "Hello, World" [SOURCE CODE]



Compiling

```
gcc -o hello hello.c
```

Note: The third word, in this case hello, is the name of the executable you are creating. This is what will be used in the next step to run the program.

Executing a program

```
./hello
```

Output

```
Hello, World!
```

Features to take note of:

1. **Comments:** comments do not affect the running of a program, and are usually there as a form of annotation to help explain or emphasize sections of code

2. **Header Files:** Headers represent the libraries that you want to include in your program, and contain code from **external** libraries for you to use. For instance, **printf** which prints statements for you is from the stdio.h library.

3. **main()**: main is the start point for all code in a C program. This is where your program starts executing. When your program is successful, you will **return 0**, and if not, you will return something else.

Note: Code wrapped in "{ }" is known as a block of code.

4. **printf()**: a function used to print formatted data onto the terminal.

5. **Statements**: All statements should end with a semicolon.

6. **Strings:** The characters enclosed in double quotes, "Hello, World\n", make up a string. This is what is printed out to the screen. The \n character is used to create a new line.

7. **Compiling**

All we have written above are instructions for our computer, to print out a statement. We did so in:

- **Source code:** A language similar to the way we normally speak and write.

However, we learnt that computers speak in terms in bytes. Therefore, we need to compile our source code into:

- **Machine code:** Our instructions represented in bytes

8. **Executing**

After compiling our code, we create a file called "hello" known as an executable. While our executable is rudimentary, merely printing a statement, many apps you see today are in fact all executables!

5. Variables & Data Types

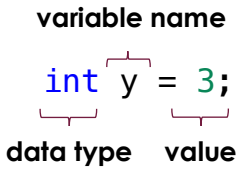
Data Types

Inevitably, you will have to work with different forms of data when programming, from numbers and decimals to characters and strings. Each of these are given a unique name and as a whole are called data types.

Variables

In computer science, variables are what programs use to store our data types and save for further use in the program. They are essentially names that we can use to access a specific data. Let's take a look at what this means in code:

Declaring a variable in C



When declaring a variable in c, there are 3 things to take note of:

1. **Data type**: In the example above, y is a variable which stores **integers**, or **int** for short. There are other types which will be explained later on.

2. **Name**: the name of the variable. This is used when you want to access the information within the variable.

3. **Value**: This is the data that the variable holds.

Example: Declaring and Initializing a Variable

```
// Format: (data type) (name of variable) = (initial value)
int main(void) {
    //declaring with no value
    int x;

    //initializing variable y with value 3
    int y = 3;
    return 0;
}
```

Note:

- In the example above, we have declared 2 variables: **x** & **y**.
- When declaring variables, you either initialize with a value, or declare its type first and assign it a value later on.
- We stored the value "3" in the variable y, but did not give x a value.

Printing variables

```
    printf("x: %d\n", x);
    printf("y: %d\n", y);
```

In order to print variables, we use what is known as a placeholder. (e.g. "`%d`").

1. When we write "%" in a string, we are telling **printf** to insert the value of our variable at that location in the string.
2. The **d** in "%d" is the type of variable we are printing. In this case, %d means to print an **integer** (int). There are other placeholders for different data types that will be explained later on.

Output

```
x: 1
y: 3
```

As you can see, we print the values of **x** & **y** respectively. However, we did not actually give x a value. In such cases, a random value will be printed by our program.

Changing the value of a variable

To change the value of a variable, we use the "**=**" operator.

```
    x = 5;
    y = 10;
    printf("x: %d\n", x);
    printf("y: %d\n", y);
```

In the code above, we are **assigning** the value 5 to x and 10 to y.

Note: The "=" should not be confused with what you learn in mathematics.

"x = 5" does not mean that x is **equals** to 5, but rather that we are **assigning** the value of 5 to x.

Output

```
x: 5
y: 10
```

As you can see, when we print x and y, we get 5 & 10 respectively!

A. Numeric Data Types

Now that we have learnt about variables and how our computer reads data, we can learn about the different **data types**. The first category would be numeric. The four fundamental types you should be familiar with are:

1. int and long**: integers**
2. float and double**: decimals**

Example of declaring and printing numeric data types

```c
int main(void) {
    int x = 3;
    long y = 100000;
    float a = 2.20;
    double b = 3.1415265359;

    printf("x: %d\n", x);
    printf("y: %ld\n", y);
    printf("a: %f\n", a);
    printf("b: %lf\n", b);
    return 0;
}
```

Output

```
x: 3
y: 100000
a: 2.200000
b: 3.141527
```

Take note of the different placeholders used to print different data types

Data Type	Placeholder
int	`%d`
long	`%ld`
float	`%f`
double	`%lf`

int vs long, float vs double

You may have noticed that longs are essentially the same as ints, and doubles are the same as floats. So why do we have these 2 data types?

In our computer, when we declare variables, we are assigning a certain amount of memory in bytes to that variable.

Data Type	Range	Space (in bytes)
int	-32,768 to 32,767	4
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8
float	$1.2E^{-38}$ to $3.4E^{38}$	4
double	$3.4E^{-4932}$ to $1.1E^{4932}$	8

As you can see, the range of a double and long is greater than that of an int and float respectively, however, they take up more space!

Thus, we are given the flexibility to choose between these data types. If we know we are not going to store a large number, then we should declare an int to save **space**. The same is true for floats and doubles.

Arithmetic Expressions

Given that we are working with numbers above, we can use arithmetic operators to manipulate the variables:

Operators in Programming

Operation	Operator (CS)
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%

```c
int main(void) {
    int x = 3;
    x = x + 5;
    printf("x + 5: %d\n", x);

    x = 10 - 2;
    printf("10 - 2: %d\n", x);

    x = 5 * 2;
    printf("5 * 2: %d\n", x);

    x = 5 / 2;
    printf("5 / 2: %d\n", x);

    x = 11 % 5;
    printf("11 mod 5: %d\n", x);
    return 0;
}
```

Output

```
x + 5: 8
10 - 2: 8
5 * 2: 10
5 / 2: 2
11 mod 5: 1
```

Note:

- When we write x = x + 5, we are **assigning** the value of "x + 1" to x. Since x was **3**, we are assigning **5 + 3**, which is **8**, into x, for the first statement.
- Modulo (%) means remainder. The remainder of **11/5** is **1**, thus **11% 5** is **1.**

Postfix expressions

Postfix Expression	Meaning
x++	x = x + 1
x−−	x = x − 1
x += x	x = x + x

In many programming languages, we can use what is known as postfix expressions to shorten arithmetic expressions

```c
int main(void) {
    int x = 3;
    x++; // x = x + 1
    printf("x: %d\n", x);

    int y = 5;
    x += y; // x = x + y
    printf("x: %d\n", x);
    return 0;
}
```

Output

```
x: 4
x: 9
```

Note: For the second print statement, x was already changed to 4 initially, thus by assigning "x + y" to x, we are assigning "4 + 5", which is **9**.

Type casting

Consider the following code:

```c
#include <stdio.h>

int main() {
    int x = 2;
    int y = 5;
    float z = (y / x);
    printf("z: %.2f\n", z);
    return 0;
}
```

Note: "%.2f" basically means print a float to 2 decimal places

Output

```
z: 2.00
```

Why is z printed as 2.00 when it is a float. Shouldn't it be printed to 2.50? This is because we are dividing 2 **ints** (x / y), which will give us another **int** (2), and then we are converting that into a **float**, hence the 2.00.

Now consider this program:

```c
int main() {
    int x = 2;
    int y =  5;
    float z = (float) y / (float) x;

    printf("z: %.2f\n", z);
    return 0;
}
```

Output

```
z: 2.50
```

By writing `(float) y`, we are telling the program to interpret y as a **float**. Now, in the above code, we are converting y & x into **floats** and therefore when we divide '5/2', we now get 2.50! This is what casting is about: converting one data type to another, and the above example is one use case of it!

B. Char Data Type

The next data type you will learn is characters (char). In C, chars are numbers from **0 – 255**, which are then converted to the corresponding character based on a conversion known as ASCII.

Data Type	Range	Space (in bytes)
int	0 - 255	1

Sample of ASCII

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
64	40	100	@	96	60	140	`
65	41	101	A	97	61	141	a
66	42	102	B	98	62	142	b
67	43	103	C	99	63	143	c
68	44	104	D	100	64	144	d
69	45	105	E	101	65	145	e
70	46	106	F	102	66	146	f
71	47	107	G	103	67	147	g
72	48	110	H	104	68	150	h
73	49	111	I	105	69	151	i
74	4A	112	J	106	6A	152	j
75	4B	113	K	107	6B	153	k
76	4C	114	L	108	6C	154	l
77	4D	115	M	109	6D	155	m
78	4E	116	N	110	6E	156	n
79	4F	117	O	111	6F	157	o
80	50	120	P	112	70	160	p
81	51	121	Q	113	71	161	q
82	52	122	R	114	72	162	r
83	53	123	S	115	73	163	s
84	54	124	T	116	74	164	t
85	55	125	U	117	75	165	u
86	56	126	V	118	76	166	v
87	57	127	W	119	77	167	w
88	58	130	X	120	78	170	x
89	59	131	Y	121	79	171	y
90	5A	132	Z	122	7A	172	z
91	5B	133	[	123	7B	173	{
92	5C	134	\	124	7C	174	|
93	5D	135	]	125	7D	175	}
94	5E	136	^	126	7E	176	~
95	5F	137	_	127	7F	177	

(image taken from https://durofy.com/ascii-values-table-generator-in-c)

If you compare the "**Dec**" and "**Char**" column, you will notice how the numbers correspond to a symbol. For instance, 65 corresponds to 'A', while 97 corresponds to 'a'. Let's take a look at this in code

Note: The image only shows ascii characters from 64 – 127.

```c
int main(void) {
    char a = 65;
    char b = 97;

    printf("a: %c\n", a);
    printf("b: %c\n", b);
    return 0;
}
```

Output

```
a: A
b: a
```

Note: The placeholder to print chars is "**%c**"

However, we don't have to write chars using their numbers, and can in fact directly write the character using single quotes (' ').

The image only shows ascii characters from 64 – 127.

```
    char a = 'A';
    char b = 'a';

    printf("a: %c\n", a);
    printf("b: %c\n", b);
```

Output

```
a: A
b: a
```

Since chars are technically numbers, we can apply arithmetic operations on them. Consider the following problem:

How do we convert a char from uppercase to lowercase? To do so, we can just add 32, because the distance between upper and lowercase letters in the ASCII table is simply 32!

```
int main(void) {
    char upper = 'C';
    char lower = upper + 32;

    printf("upper: %c\n", upper);
    printf("lower: %c\n", lower);
    return 0;
}
```

Output

```
upper: C
lower: c
```

Note: Words and **sentences** are a data type called **string**. However, in C, there is no actual data type called string, but rather these are a special case of **char** that you will learn about later on.

C. Bool

Perhaps the simplest data type of all is bool (stands for Boolean), and with this you are able to assert a value of true or false.

Data Type	Range	Space (in bytes)
bool	true or false	1

Bools are technically numbers as well. It is:

- 1 if **true**
- 0 or otherwise if **false**

```c
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool isTrue = true;
    bool isFalse = false;

    printf("true: %d\n", isTrue);
    printf("false: %d\n", isFalse);
    return 0;
}
```

Output

```
true: 1
false: 0
```

Note:

- To use bools in C, we have to include the **stdbool.h** library.
- There are no placeholders to print bools. We have to simply use %d.

Use case of Booleans:

On many applications, we have switches which toggle an on and off state. Most of these are implemented through the use of Booleans.

6. Reading Input

A key component of programming is being able to take an input from a user, storing it in a variable, and manipulating it.

There are many ways to do so in C. For now, we will use the **`scanf`** *function*. In order to use this function, you must:

1. Declare a variable in which to store the input, then
2. Call scanf to store input into that variable

```c
int main(void) {
    int myAge;

    printf("What is your age?\n" );
    //    syntax for scanf:
    //    scanf((placeholder), &(variable name))
    scanf("%d", &myAge);
    printf("You are %d years old\n", myAge);
    return 0;
}
```

Output

```
What is your age?
20
You are 20 years old
```

The syntax for scanf is such that you must have:

1. The placeholder of the data type being read from the input stream.

2. The variable name in the form "`&(variable name)`". & stands for address which you will learn about later on.

Note:

- A function is essentially a task that we want to perform

- The placeholder and variable name are called arguments that you pass into a function. This will be taught in the next unit on functions.

7. Expressions

An expression (or statement) is a line of code that may be evaluated to result in a value. You already learnt about arithmetic expressions. Now, you will learn another type, which is conditional expressions.

Conditional expressions

In conditions, you write a block of code (code wrapped in "{ }") which runs only if your statement is true. Conditional statements essentially equate to bool values. The code below is equivalent to "**if x is less than y, run code within block**".

```
    if (x < y) {
        //code within this block will run if statement is true
    }
```

Comparison	Meaning
==	Equals to
>	Greater than
<	Less than
>=	Greater than equals to
<=	Less than equals to

```
int main(void) {
    int x;
    printf("x: ");
    scanf("%d", &x);

    if (x < 5) {
        printf("x is less than 5\n");
    }
    return 0;
}
```

Output 1

```
x: 3
x is less than 5
```

Output 2

```
x: 5
```

Notice how on the first run of the code, we set x to be **3**, which is less than 5, and therefore the statement was printed. However, on the second run, we set x to be 5, and thus no statement was printed.

If & Else

Your code will run an else block if the initial **if** statement is not true.

```c
int main(void) {
    int x = 3;
    int y = 5;

    if (x == y) {
        printf("x is equals to y\n");
    }
    else {
        printf("x is not equals to y\n");
    }
    return 0;
}
```

Output

```
x is not equals to y
```

Else if

You can check all cases of a truth statement using the else if statement.

```c
int main(void) {
    int x = 3;
    int y = 5;

    if (x == y) {
        printf("x is equals to y\n");
    }
    else if (x > y){
        printf("x is greater than y\n");
    }
    else if (x < y) {
        printf("x is less than y\n");
    }
    return 0;
}
```

Output

```
x is less than y
```

Writing good code: Instead of the above code, how can you change it to make it such that the code is neater?

Logical Operators

When we want to test for multiple truth statements, we use conditional operators.

Logical Operation	Operator (CS)
AND	&&
OR	||

```
int main(void) {
    int x = 3;
    int y = 5;

    //  code will run if both statements are true
    if (x == 3 && x < y) {
        printf("x is equals to 3 and less than y\n");
    }
    //  code will run if one or the other statement is true
    if (y == 5 || y < 3){
        printf("y is equals to 3 or less than 5\n");
    }
    return 0;
}
```

Output

```
x is equals to 3 and less than y
y is equals to 3 or less than 5
```

8. Loops

Loops are used in programming to repeat a block of code until a specified condition is met. It is a method of iteration. **Iteration** is a key concept in computer science, which means to repeat a process or function, allowing you to manipulate data within that process (loop) until an endpoint is reached.

While Loops

While loops evaluate a test expression wrapped in parenthesis and runs the block of code until the expression is evaluated to false.

```c
int main(void) {
    int x = 0;
    while(x < 5) {
        printf("x is %d\n", x);
        x++;
    }
    return 0;
}
```

Output:

```
x is 0
x is 1
x is 2
x is 3
x is 4
```

Do … While Loops

Do … While loops are similar to while loops, except that the block of code runs first, before evaluating the test expression.

```c
int main(void) {
    int x = 10;
    do {
        printf("x is %d\n", x);
        x++;
    }
    while(x < 10);
    return 0;
}
```

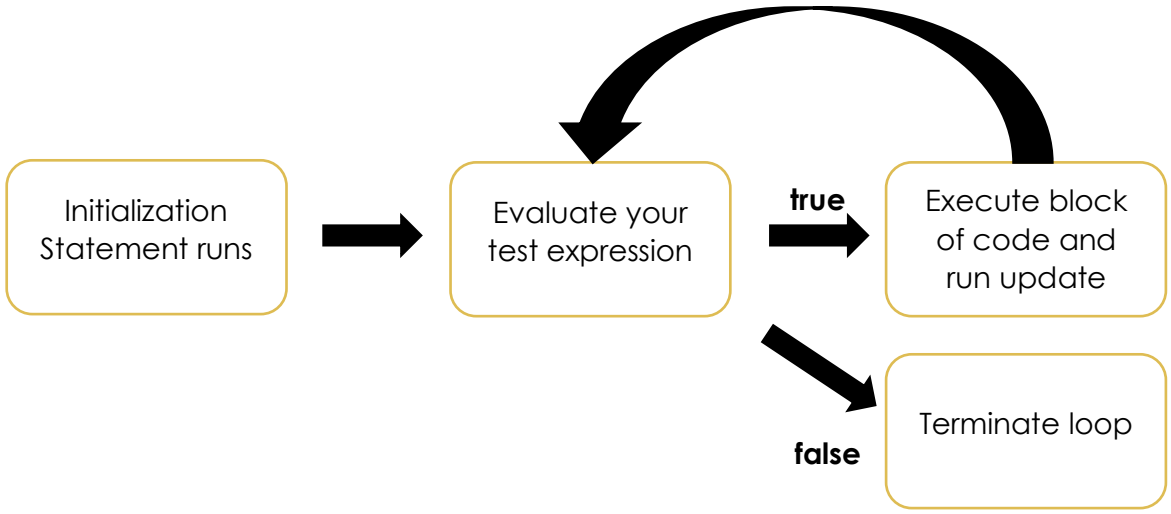
Output:

```
x is 10
```

For Loops

The way that for loops work is that you

1. Provide it an initialization statement
2. Provide a test expression
3. Provide an update statement




```
for (int i = 0; i < 5; i++ ) {
    printf("i is %d\n", i);
}
```

initialization — test — update — body

Example of for loop

```
int main(void) {
    for (int i = 0; i < 5; i++ ) {
        printf("i is %d\n", i);
    }
    return 0;
}
```

Output:

```
i is 0
i is 1
i is 2
i is 3
i is 4
```

The output is the same as before, but the method used is different. Usually, we use for loops when we know how many time our loop will take place (number of iterations), and while loops when we are unsure of when the loop will end.

Nesting For Loops

Nesting for loops is a common method seen in programming. For instance, if you wanted to build a 2-d output (rows and columns), you can do so through nesting.

```c
int main(void) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%d  ", i);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
0  0  0  0  0
1  1  1  1  1
2  2  2  2  2
3  3  3  3  3
4  4  4  4  4
```

9. Variable Scope

A variable's scope refers to the block of code in a program in which it is available in. When you create a variable within a block, it's scope consists of the block it was created in, as well as any nested blocks within.

The key idea here is that:

Within a block, you can only access variables that have been initialized within that block and outer scoped blocks, but not vice versa.

```c
int main(void) {
    int x = 10;
    i = x + 3;
    for (int i = 0; i < 5; i++ ) {
        i++;
        printf("i is %d", i);
    }
    return 0;
}
```

Take a look at the following example. If you try to compile it, you will receive the following error.

Output:

```
main.c:7:5: error: use of undeclared identifier 'i'
    i = x + 3;
    ^
1 error generated.
```

This is because **i** was initialized in an inner scope block.