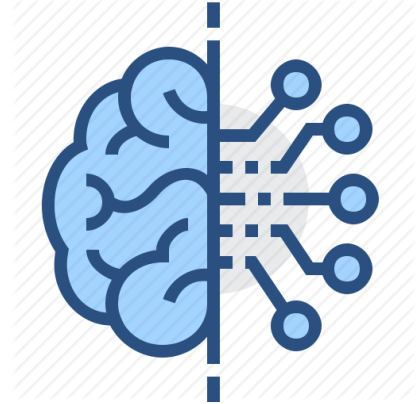


# Thinking Machine - 2



## How to Read This Book

This book is not about teaching, it's more about exploring. Author is noob trying to learn and know how computer works. Also, he wants to make projects to impress his crush on her birthday.

All the topics are divided into 3 major parts:

1. Why (Why is it important for author to read this new topic).
2. How (How will this topic help author in making effective project).
3. What (What is in the topic to learn).

**Complete this book, take free course at:**

<https://electroverlabs.com/courses/introduction-to-c-programming/>

Give quiz and download **Completion certificate**. Passing marks 75%

**Special Thanks to:**

**Authors of blogs written on different topics.**

**By**

*(This book is compiled by Electroverts lab)*

## Before starting, author wants to convey some message:

- It's totally okay if you don't know anything and completely new to Computers and Arduino.  
The fact that you are reading this makes you enter the group of people who wants to explore possibilities of the new world where everyone gets chance to do something meaningful and productive to change the world for good.
- Author is complete noob (new comer) in this world of computers. He don't want to teach anybody anything. He is just sharing whatever he is exploring and trying to make it simple.
- This book is not written for exam point of view, instead here author wants to explain how to think.
- Practice 5 Why concept. To dig deep in any topic, keep asking why, why 5 times and you'll know root cause.
- There is no point of memorizing anything, consider this book as a story of Computers.
- If you feel something is important, repeat it one more time.
- **Learn the art of using google, it will help you a lot to clear your concepts and get answers of different questions going in your mind.**
- **Learn the art of using google, it will help you a lot to clear your concepts and get answers of different questions going in your mind.**
- Try not to skip theory part as knowing what you are using (computer) will give you visibility on what exactly is happening when you write codes or use Instagram.
- Try to maintain consistency throughout, learning how to write code is like playing video games. You'll easily get bored and feel weak in beginning and with time you'll be able to think logically on your own and deploy strategies to conquer.

## Table of Contents

Table of Contents .....	iii
Understanding Programming .....	8
WHAT IS THE DIFFERENCE BETWEEN CODING & PROGRAMMING? .....	8
How Coding Works? .....	8
Why Do We Have So Many Languages? .....	9
Low-Level and High-Level Languages .....	9
Programs! .....	10
What Happens When You Run a Program? .....	10
What Is an Algorithm? .....	11
Compiler? .....	11
Interpreter? .....	12
Online Compilation! .....	12
Let's learn a new language .....	13
1. Computer Programming – Environment .....	13
SYNTAX HIGHLIGHTING .....	13
AUTOCOMPLETE .....	14
DEBUGGING .....	15
2. Basic Syntax .....	16
1. #include <stdio.h> .....	17
2. void main() .....	17
3. printf("Hello World"); .....	18
4. Comments .....	18
5. Whitespaces .....	18
6. Semicolons .....	19
7. Syntax Error .....	19
3. Data Types .....	20
4. Variables .....	22
printf(); in C .....	22
5. Keywords .....	24
6. BasicOperators .....	25
Arithmetic Operators: .....	25
Relational Operators: .....	26

Logical Operators:.....	26
Assignment Operators: .....	27
Increment/Decrement Operators:.....	28
Bitwise Operators: .....	29
7. Decision Making.....	30
1. if statement in C .....	31
2. if-else in C.....	32
3. nested-if in C .....	33
4. if-else-if ladder in C .....	33
5. Jump Statements in C .....	34
6. Switch Statement .....	38
7. Ternary Operator .....	39
8. Loops .....	41
1. while loop in C .....	42
2. for Loop .....	42
3. do while loop in C .....	44
scanf().....	45
9. Numbers .....	48
Math Operations on Numbers .....	49
10. Characters .....	51
Escape Sequences .....	52
11. Arrays.....	54
Accessing Array Elements:.....	56
Some facts on array:.....	57
12. Strings .....	59
13. Functions .....	61
1) Predefined standard library functions .....	62
2) User Defined functions .....	64
14. Miscellaneous .....	72
1. floor() and ceil() functions of math.h in C .....	72
2. Typcasting in C .....	73
Implicit Type Conversion .....	73
Explicit Type Conversion.....	74
3. Prefix and Postfix operators in C .....	75

4. Hierarchy of operators and their associativity .....	76
5.Scope of Variables in C .....	77
Local Variables .....	77
Local Scope .....	77
Global Variables .....	78
Global Scope .....	78
6. Headers in C .....	78
List of commonly used header files in C .....	79
7. Macros using #define .....	79
8. Static Variables in C .....	80
9. Two Dimensional Arrays in C .....	80
10. IDE vs Editor .....	81
11. <ctype.h> .....	81
C isalnum() .....	81
C isalpha() .....	81
C isdigit() .....	81
C isgraph() .....	81
C islower() .....	81
C isprint() .....	81
C ispunct() .....	81
C isspace() .....	81
C isupper() .....	81
C tolower() .....	81
C toupper() .....	82
12.The Comma Operator .....	82
Let's Code .....	84
1. Installing compiler, IDE & Editor .....	84
Code: Basic Programs .....	87
1. Hello World Program .....	87
2. Program to take input of various datatypes in C .....	87
3. C Program to Add Two Integers .....	88
4. C Program to Multiply Two Floating-Point Numbers .....	90
5. C Program to Find ASCII Value of a Character .....	91

6. C Program to Compute Quotient and Remainder .....	92
7. C Program to Find the Size of int, float, double and char .....	93
8. C Program to Swap Two Numbers .....	94
9. C Program to Check Whether a Number is Even or Odd .....	95
10. C Program to Check Whether a Character is a Vowel or Consonant .....	96
11. C Program to Find the Largest Number Among Three Numbers .....	98
12. C Program to Find the Roots of a Quadratic Equation .....	101
13. C Program to Check Leap Year .....	102
14. C Program to Check Whether a Number is Positive or Negative .....	103
15. C Program to Check Whether a Character is an Alphabet or not .....	105
16. C Program to Calculate the Sum of Natural Numbers .....	106
17. C Program to Find Factorial of a Number .....	108
18. C Program to Generate Multiplication Table .....	109
19. C Program to Display Fibonacci sequence .....	111
20. C Program to Display Characters from A to Z Using Loop .....	112
21. C Program to Count Number of Digits in an Integer .....	114
22. C Program to Reverse a Number .....	115
23. C Program to Calculate the Power of a Number .....	116
24. C Program to Check Whether a Number is Palindrome or Not .....	117
25. C Program to Check Whether a Number is Prime or Not .....	118
26. C Program to Display Prime Numbers between Two Intervals .....	120
27. C Program to Check Armstrong Number .....	122
28. C Program to Display Factors of a Number .....	125
29. C Program to Find GCD of two Numbers .....	126
30. C Program to Find LCM of two Numbers .....	129
31. C Program to Print Pyramids and Patterns .....	131
32. C program to build a Basic Calculator .....	138
EXAMPLE PROGRAMS USING FUNCTIONS .....	140
33. C program to sum each digit of a number .....	140
34. C Program to find the factorial of a number using user defined functions .....	141
35. C program to check whether a number is palindrome or not using user defined functions .....	142
36. C program to check if the number is prime or not using user defined functions .....	144
EXAMPLE PROGRAMS ON 1-D ARRAYS .....	145
37. C program to perform Simple Linear Search .....	145

38. C program to sort elements of an array using bubble sort.....	146
EXAMPLE PROGRAMS ON STRINGS .....	148
39.C program to find the length of given string without using string functions .....	148
40. C program to find the number of vowels, consonants and white spaces in a given string .....	149
Additional Content .....	150
1. How does a C program executes? .....	150
2. When you are installing a program, are you really just compiling it? .....	152
3. How is a programming language created and developed? .....	152
4. Why Are There So Many Programming Languages? .....	154
5. History of C Language.....	156

# Understanding Programming

## WHAT IS THE DIFFERENCE BETWEEN CODING & PROGRAMMING?

Coding involves writing many lines of code in order to create a software program. Programming involves not only coding but also other tasks, such as analyzing and implementing algorithms, understanding data structures, solving problems, and more. Programmers are typically technically-minded and have strong analytical skills.

So basically saying, coding is a subset of programming. Although we will use both words referring to same meaning, so don't be confused.

## How Coding Works?

So how does coding work, really? The short answer is that writing code tells the computer what to do, but it's not quite that simple.

So here's the longer answer. A computer can only understand two distinct types of data: on and off. In fact, a computer is really just a collection of on/off switches (transistors). Anything that a computer can do is nothing more than a unique combination of some transistors turned on and some transistors turned off.

Binary code is the representation of these combinations as 1s and 0s, where each digit represents one transistor. Binary code is grouped into bytes, groups of 8 digits representing 8 transistors. For example, 11101001. Modern computers contain millions or even billions of transistors, which means an unimaginably large number of combinations.

But one problem arises here. To be able to write a computer program by typing out billions of 1s and 0s would require superhuman brainpower, and even then it would probably take you a lifetime or two to write.

This is where programming languages come in...

```
print('Hello, world!')
```

That line of code is written in the Python programming language. Put simply, a programming (or coding) language is a set of syntax rules that define how code should be written and formatted.

Thousands of different programming languages make it possible for us to create computer software, apps and websites. Instead of writing binary code, they let us write code that is (relatively) easy for us to write, read and understand. Each language comes with a special program that takes care of translating what we write into binary code.



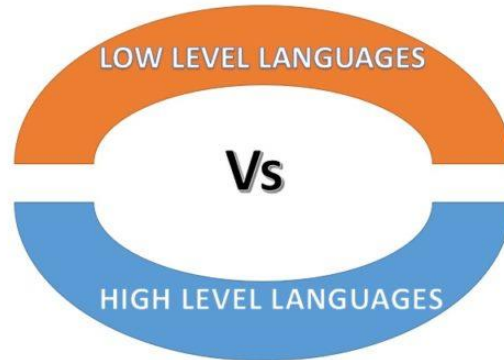
## Why Do We Have So Many Languages?

Because different languages are designed to be used for different purposes – some are useful for web development, others useful for writing desktop software, others useful for solving scientific and numeric problems, and so on.

## Low-Level and High-Level Languages

Programming languages can also be low-level or high-level.

Low-level languages are closer to the binary code a computer understands, while high-level languages bear a lot less resemblance to binary code. High-level languages are easier to program in, because they're less detailed and designed to be easy for us to write.



Nearly all of the main programming languages in use today are high-level languages.

S.NO	HIGH LEVEL LANGUAGE	LOW LEVEL LANGUAGE
1.	It is programmer friendly language.	It is a machine friendly language.
2.	High level language is less memory efficient.	Low level language is high memory efficient.
3.	It is easy to understand.	It is tough to understand.
4.	It is simple to debug.	It is complex to debug comparatively.
5.	It is simple to maintain.	It is complex to maintain comparatively.
6.	It is portable.	It is non-portable.

7.	It can run on any platform.	It is machine-dependent.
8.	It is used widely for programming.	It is not commonly used now-a-days in programming.
9.	It needs compiler or interpreter for translation.	It needs assembler for translation.

## Programs!

A program is simply a text file, written in a certain coding language. The code inside a program file is called the source code. Every coding language has its own file extension for identifying code files written in that language. For example, Python's is '.py', C language have extension as '.c'.

To make a program, you write the code in a plain text editor like Notepad and save the file to your computer. That's it. For example, the below line of code could be the contents of a very short Python program called hello.py:

```
print('Hello, world!')
```

How do you run a program and actually get it to perform its commands? That varies between coding languages. Some languages save a separate binary file that the computer can directly run, while other languages have their programs run indirectly by certain software.

For example, a JavaScript & HTML program files would get run by a web browser like Chrome. A PHP program file would get run by a web server like XAMP.

In the case of our hello.py file, we will see output on terminal/command prompt. Same in case of other languages like C, C++, JAVA, RUBY etc.

## What Happens When You Run a Program?

A computer doesn't actually understand the phrase 'Hello, world!', and it doesn't know how to display it on screen. It only understands on and off. So to actually run a command like `print('Hello, world!')`, it has to translate all the code in a program into a series of ons and offs that it can understand.

### **To do that, a number of things happens:**

1. The source code is translated into assembly language.
2. The assembly code is translated into machine language.
3. The machine language is directly executed as binary code.

Confused? Let's go into a bit more detail. The coding language first has to translate its source code into assembly language, a super low-level language that uses words and numbers to represent binary patterns. Depending on the language, this may be done with an interpreter (where the program is translated line-by-line), or with a compiler (where the program is translated as a whole).

[Author: printf\("Electroverts Lab"\);](#)

The coding language then sends off the assembly code to the computer's assembler, which converts it into the machine language that the computer can understand and execute directly as binary code.

Which then finally executes showing us the output we desired... i.e. `print("Hello World")` in python will print **Hello World** when the source code will be compiled/interpreted and computer assembler will then convert assembly code came as output from compiler/interpreter to binary code which finally runs and our machine understands it.

## What Is an Algorithm?

From programming point of view, an algorithm is a step-by-step procedure to resolve any problem. An algorithm is an effective method expressed as a finite set of well-defined instructions.

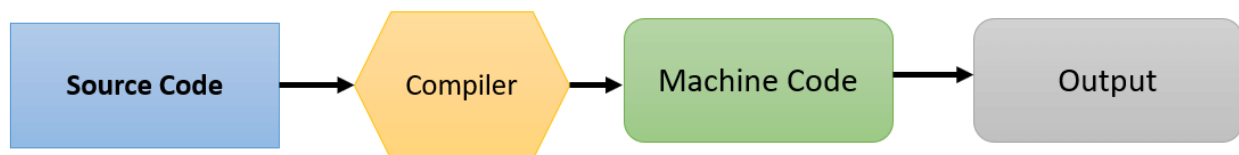
Thus, a computer programmer lists down all the steps required to resolve a problem before writing the actual code.

## Compiler?

You write your computer program using your favorite programming language and save it in a text file called the program file.

Now let us try to get a little more detail on how the computer understands a program written by you using a programming language. Actually, the computer cannot understand your program directly given in the text format, so we need to convert this program in a binary format, which can be understood by the computer.

### How Compiler Works



The conversion from text program to binary file is done by another software called Compiler and this process of conversion from text formatted program to binary format file is called program compilation. Finally, you can execute binary file to perform the programmed task.

We are not going into the details of a compiler and the different phases of compilation.

So, if you are going to write your program in any such language, which needs compilation like C, C++, Java and Pascal, etc., then you will need to install their compilers before you start programming.

[Author: printf\("Electroverts Lab"\);](#)

## Interpreter?

We just discussed about compilers and the compilation process. Compilers are required in case you are going to write your program in a programming language that needs to be compiled into binary format before its execution.

### How Interpreter Works



There are other programming languages such as Python, PHP, and Perl, which do not need any compilation into binary format, rather an interpreter can be used to read such programs line by line and execute them directly without any further conversion.

So, if you are going to write your programs in PHP, Python, Perl, Ruby, etc., then you will need to install their interpreters before you start programming.

## Online Compilation!

Online compilers are tools which allows you to compile source code and execute it online in many programming languages.

Try this one: <https://repl.it/>

## Let's learn a new language

We assume you are well aware of English Language, which is a well-known Human Interface Language. English has a predefined grammar, which needs to be followed to write English statements in a correct way. Likewise, most of the Human Interface Languages (Kannada, Tamil, Telugu, English, Hindi, Spanish, French, etc.) are made of several elements like verbs, nouns, adjectives, adverbs, propositions, and conjunctions, etc.

Similar to Human Interface Languages, Computer Programming Languages are also made of several elements. We will take you through the basics of those elements and make you comfortable to use them in various programming languages. These basic elements include –

1. Programming Environment
2. Basic Syntax
3. Data Types
4. Variables
5. Keywords
6. Basic Operators
7. Decision Making
8. Loops
9. Numbers
10. Characters
11. Arrays
12. Strings
13. Functions

### 1. Computer Programming – Environment

Now that you are ready to start programming, let's find us a platform to do it.

You can do programming in Notepad, yes you heard it correct. Although for making things easy for us, people have developed IDE (Integrated Development Environment) because IDE gives us some additional features which makes writing codes easy and interesting.

IDE is nothing but an editor software like notepad which have some additional features like:

#### SYNTAX HIGHLIGHTING

An IDE/Editor that knows the syntax of your language can provide visual cues. Keywords, words that have special meaning like class in Java, are highlighted with different colors.

[Author: printf\("Electroverts Lab"\);](#)

```
// without syntax highlighting

public class NiceDay {
    public static void main(String[] args) {
        System.out.println("It's a nice day out!");
    }
}
```

```
// with syntax highlighting

public class NiceDay {
    public static void main(String[] args) {
        System.out.println("It's a nice day out!");
    }
}
```

Syntax highlighting makes code easier to read by visually clarifying different elements of language syntax.

## AUTOCOMPLETE

When the EDITOR/IDE knows your programming language, it can anticipate what you're going to type next!

We've seen statements with `System.out.println()` quite a bit so far. In an IDE, we might see `System` as an autocomplete option after only typing `Sy`. This saves keystrokes so the programmer can focus on logic in their code.

```
public static void main(String[] args) {
    Sy
}
System (java.lang)
SynchronousQueue<E> (java.util
```

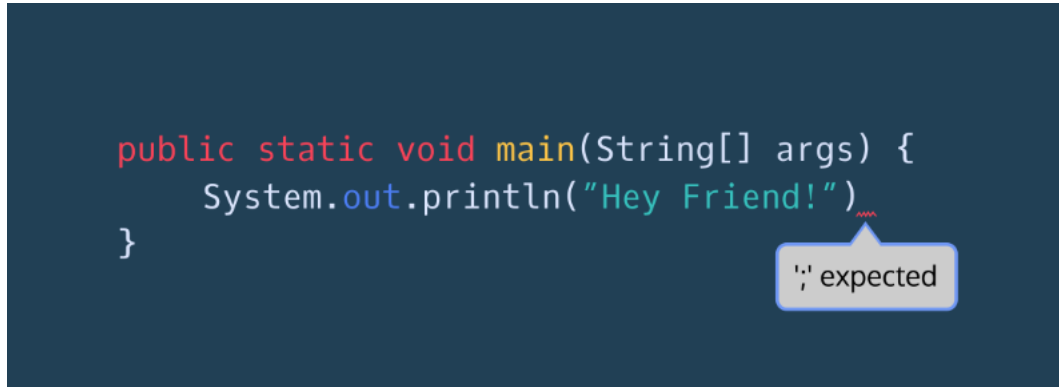
[Author: printf\("Electroverts Lab"\);](#)

## **DEBUGGING**

No programmer avoids writing bugs and programs with errors.

When a program does not run correctly, EDITOR/IDEs provide debugging tools that allow programmers to examine different variables and inspect their code in a deliberate way.

EDITOR/IDEs also provide hints while coding to prevent errors before compilation.



**I use mainly 2 EDITOR/IDEs for C and C++ programming.**

Visual Studio Code: <https://code.visualstudio.com/download>

Code Blocks: <https://sourceforge.net/projects/codeblocks/files/Binaries/20.03/Windows/codeblocks-20.03-setup.exe/download>

## 2. Basic Syntax

Let's start with a little coding, which will really make you a computer programmer. We are going to write a single-line computer program to write Hello, World! On your screen. Let's see how it can be written using different programming languages.

### Hello World Program in C:

```
#include <stdio.h>

void main() {

    /* printf() function to write Hello, World! */

    printf( "Hello, World!" );

}
```

```
# print function is used to print on
screen

print("Hello World!")
```

### Hello World Program in Python:

Which produces the following result –

### Hello, World!

Yes, you have to write more on C as compared to Python for achieving same result most of the times. That's because Python is developed much after C and is high level language while C is mid-level, meaning its closer to machine level as compared to python and is faster as compared to python.

As a newcomer, we start with C programming though it is not the easiest language but it is simpler as we follow Imperative programming paradigm in C where we describe sequence of steps and our machine follows it while in Python and other high level languages we follow different more complicated paradigm for different purposes.

**We are keeping our primary language as C. For our purpose C will basically help us in programming Arduino Boards.**

**We will also realize that every programming language have very similar concept, if we understand 1 programming language it will be very easy for us to learn others.**

Now coming back to our C code for printing "Hello World!"

Let us understand what is happening there.

1. I wrote the code in C language.
2. I saved it and then opened cmd(command) prompt/terminal & changed my directory (folder) to the file path where I saved my C code.
3. Then I called compiler and compiled my code, which eventually made a ".exe" file in same directory.
4. Finally, I ran my ".exe" file to see my output.

"Yes, you made your first software here".



You are basically making a software that prints Hello World! On screen whenever it will run.  
Let's take a step back and try to understand our C code in little detail.

**1. #include <stdio.h>** >> In C programming, header files provides a way to reach library code. (confused? Let's make it simple)

In the code above we used "printf" statement which actually prints our string of words on screen. Question is where does "printf" comes from and who wrote it such that whenever we put something inside its braces, it will print that on screen?

Answer is Mike Lesk (American computer scientist) wrote that piece of code which will take input inside "printf" function and will show that on output screen. Now that he wrote it for himself, he then wanted to make it available to other users and thus he saved his code as a library file.

Library are a package of code which is reused many times and that code is precompiled, hence it is available in standard form so that we do not have to write that code for every program that we develop. And header file contains the reference to that code in simple way the functions we use in our program like "scanf" and "printf" are fully defined in a standard library and header files like **stdio.h** header file contains the reference to that code. So when we compile our code, we just get the precompiled for scanf and printf, and we do not have to write the code for scanf and printf every time we use it. In simpler way we can say that a library contains codes for all the functions and a header file is way to reach that code.

We first import the **stdio.h** library (the name stands for standard input-output library).

This library gives us access to input/output functions like scanf, printf etc.

C is a very small language at its core, and anything that's not part of the core is provided by libraries. Some of those libraries are built by normal programmers, and made available for others to use. Some other libraries are built into the compiler. Like stdio and others.

This function is wrapped into a main() function. The main() function is the entry point of any C program..

**2. void main()** >> main() function is the entry point of any C program. It is the point at which execution of program is started. When a C program is executed, the execution control goes directly to the main() function. Every C program have a main() function.

Functions are small units of programs and they are used to carry out a specific task. For example, the above program makes use of two functions: main() and printf(). Here, the function main() provides the entry point for the program execution and the other function printf() is being used to print an information on the computer screen.

You can write your own functions, but C programming itself provides various built-in functions like main(), printf(), etc., which we can use in our programs based on our requirement.

"Void" here means that main() function is returning nothing. We will understand [functions](#) later in this module and then we will properly understand what is "void" written there!

**3. [printf\("Hello World"\);](#)** >> This function displays the content within double quotes as it is on the screen.

**4. [Comments](#)** >> A C program can have statements enclosed inside `/*.....*/`. Such statements are called comments and these comments are used to make the programs user friendly and easy to understand. The good thing about comments is that they are completely ignored by compilers and interpreters. So you can use whatever language you want to write your comments.

`/* printf() function to write Hello, World! */` in C for multiline

or

`// printf() function to write Hello, World!`  in C for single line

**5. [Whitespaces](#)** >> When we write a program using any programming language, we use various printable characters to prepare programming statements. These printable characters are **a, b, c,.....z, A, B, C,.....Z, 1, 2, 3,..... 0, !, @, #, \$, %, ^, &, \*, (, ), -, \_, +, =, \, |, {, }, [, ], :, ;, <, >, ?, /, \, ~, `., ', . etc.**

Apart from these characters, there are some characters which we use very frequently but they are invisible in your program and these characters are spaces, tabs (`\t`), new lines(`\n`). These characters are called whitespaces.

These three important whitespace characters are common in all the programming languages and they remain invisible in your text document –

Whitespace	Explanation	Representation
<b>New Line</b>	To create a new line	<code>\n</code>
<b>Tab</b>	To create a tab.	<code>\t</code>
<b>Space</b>	To create a space.	empty space

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it. Whitespace is the term used in C to describe blanks, tabs, newline characters, and comments. So you can write `printf("Hello, World!" );` as shown below. Here all the created spaces around "Hello, World!" are useless and the compiler will ignore them at the time of compilation.

```
#include <stdio.h>

void main() {

    /* printf() function to write Hello, World! */

    printf( "Hello, World!" );

}
```

which produces the following result –

Hello, World!

**6. Semicolons** >> Every individual statement in a C Program must be ended with a semicolon (;), for example, if you want to write "Hello, World!" twice, then it will be written as follows –

```
#include <stdio.h>

void main() {

    /* printf() function to write Hello, World! */

    printf( "Hello, World!" );

    printf( "Hello, World!" ); }
```

This program will produce the following result –

Hello, World!Hello, World!

But we want to see our output in 2 different lines, for that we have to put a newline character at the end. Check the updated code below –

```
#include <stdio.h>

void main() {

    /* printf() function to write Hello, World! */

    printf( "Hello, World! \n" );

    printf( "Hello, World!" ); }
```

This program will produce the following result –

Hello, World!

Hello, World!

**7. Syntax Error** >> If you do not follow the rules defined by the programming language, then at the time of compilation, you will get syntax errors and the program will not be compiled. From syntax point of view, even a single dot or comma or a single semicolon matters and you should take care of such small syntax as well. In the following example, we have skipped a semicolon, let's try to compile the program –

Missing semicolon

```
#include <stdio.h>

void main() {

    /* printf() function to write Hello, World! */

    printf( "Hello, World!" )

}
```

This program will produce the following result –

main.c: In function 'main':

main.c:7:1: error: expected ';' before '}' token.

### 3. Data Types

Let's discuss about a very simple but very important concept available in almost all the programming languages which is called data types. As its name indicates, a data type represents a type of the data which you can process using your computer program. It can be numeric, alphanumeric, decimal, etc.

Let's take another example where we want to record student information in a notebook. Here we would like to record the following information –

Name:  
Class:  
Section:  
Age:  
Sex:

Now, let's put one student record as per the given requirement –

Name: Alyusha  
Class: 6th  
Section: J  
Age: 13  
Sex: F

- Student name "Alyusha" is a sequence of characters which is also called a string.
- Student class "6th" has been represented by a mix of whole number and a string of two characters. Such a mix is called alphanumeric.
- Student section has been represented by a single character which is 'J'.
- Student age has been represented by a whole number which is 13.
- Student sex has been represented by a single character which is 'F'.

This way, we realized that in our day-to-day life, we deal with different types of data such as strings, characters, whole numbers (integers), and decimal numbers (floating point numbers).

Similarly, when we write a computer program to process different types of data, **we need to specify its type clearly**; otherwise the computer does not understand how different operations can be performed on that given data. Different programming languages use different keywords to specify different data types. For example, C and Java programming languages use int to specify integer data, whereas char specifies a character data type.

Following are the examples of some very common data types used in C:

**char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.

**int:** As the name suggests, an int variable is used to store an integer.

**float:** It is used to store decimal numbers (numbers with floating point value) with single precision.

**double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler(we will be using same compiler).

DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	8	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	8	0 to 4,294,967,295	%lu
long long int	8	-(2 <sup>63</sup> ) to (2 <sup>63</sup> )-1	%lld
unsigned long long int	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	16		%Lf

We can calculate how much range can be there for each data type as their size is already given.

For example, **char** takes 1 byte of data.

1 byte = 8 bits 

We can store 2 things in each bit either 0 or 1 (off or on).

That makes a total possibility of storing 2<sup>8</sup> different combinations, so we see 256 combination of 0s and 1s here where each combination refers to certain character. (2<sup>8</sup> = 256)

Basically we can use this 1 byte to store all different types of **characters** (not string) like a, Z, !, @ etc.

We have lesser character in English than 256. Hence we coded in our compiler that if 8-digit sequence of certain number comes, consider it as "A" or "#" etc.

So we fixed already that whenever **char** datatype will be called, it will take not more than 1 byte of memory.

Coming down to **int**, which takes 4 byte of data.

1 byte = 8 bits, 4 byte = 32 bits.

So we can eventually store 2<sup>34</sup> = 4,294,967,296 combinations in **int** data type. If our number exceeds that, we have to store that in **long, double or long double** depending on the size of data.

If you want to understand how 1 is written in binary refer to this link:

<https://www.youtube.com/watch?v=rsxT4FfRBaM>

<https://owlcation.com/stem/How-to-Convert-Decimal-to-Binary-and-Binary-to-Decimal>

Now let's understand what "signed" and "unsigned" means.

Unsigned means that number will be positive or 0 only. So, in case of **int** datatype, all 2<sup>32</sup> combinations will give positive number only and that ranges from 0 to 4,294,967,295.

In case of signed **int**, 1<sup>st</sup> bit is used to specify if the number is +ve or -ve and remaining 31 bits are used to make different combinations.

Thus 2<sup>31</sup> = 2147483648 is total combination and 1<sup>st</sup> bit will then give it status of -ve or +ve.

So, signed **int** will range from -2,147,483,648 to 2,147,483,647.

In simple words signed are used when there is a possibility of negative numbers too.

## 4. Variables

A variable in simple terms is a storage place which has some memory allocated to it. Basically, a variable is used to store some form of data. Different types of variables require different amounts of memory, and have some specific set of operations which can be applied on them. It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
int a;
```

```
float b;
```

```
char c;
```

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

```
int a=10,b=20;//declaring 2 variable of integer type
```

```
float f=20.8; // int datatype is used to store integer values only while float is used to store real numbers which includes decimals. So, for storing any real number we can use float.
```

```
char c='A';
```

So basically, we are telling compiler that we will be storing an integer type value i.e. 10 in case above and we will name the storage where 10 is stored as "a", so that whenever "a" is called it will throw out stored value in that location that is 10.

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names:

```
int a;
```

```
int _ab;
```

```
int a30;
```

Invalid variable names:

```
int 2; // can't start with a number
```

```
int ab; // no space
```

```
int long; // no keyword
```

## printf(): in C

Problem: I want to add 2 numbers and print that on screen.

```
Solution:      #include <stdio.h>
                void main() {
                  int a = 12, b=16;
                  printf("%d \n", a+b);
                }
```

It will give output as >>> 28

Wonder what on earth are those strange characters? i.e. %d and \n.

[Author: printf\("Electroverts Lab"\);](#)

In C programming language, printf() function is used to print the ("character, string, float, integer, octal and hexadecimal values") onto the output screen.

We use printf() function with %d format specifier to display the value of an integer variable (**int**). Similarly, %c is used to display character (**char**), %f for **float** variable, %s for **string** variable & %lf for **double**. ([format specifier](#))

To generate a newline, we use "\n" in C printf() statement. A newline format specifier will basically create a new line so that next characters are printed on that new line.

If we write printf("a \nb"); it will give following output.



And if we write printf("a \nb"); it will give following output.



That difference came because we put space after the escape sequence. There are many different types of escape sequences. We are going to study about them in coming topics.

## 5. Keywords

So far, we have covered two important concepts called variables and their data types. We discussed how to use int, char, and float to specify different data types. We also learnt how to name the variables to store different values.

Though this topic is not required separately because reserved keywords are a part of basic programming syntax, we kept it separate to explain it right after data types and variables to make it easy to understand.

Like int, char, and float, there are many other keywords supported by C programming language which we will use for different purpose. Different programming languages provide different set of reserved keywords, but there is one important & common rule in all the programming languages that **we cannot use a reserved keyword to name our variables**, which means we cannot name our variable like int or float rather these keywords can only be used to specify a variable data type.

### C Programming Reserved Keywords

Here is a table having almost all the keywords supported by C Programming language –

You don't need to memorize any of this.

Just know that keywords exists and you can't name your variables with these names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			



## 6. Basic Operators

Operators are symbols that tells the compiler to perform specific mathematical or logical manipulations. We will try to cover the most commonly used operators in programming.

First, let's categorize them:

1. Arithmetic
2. Relational
3. Logical
4. Assignment
5. Increment
6. Bitwise

### Arithmetic Operators:

Arithmetic operators are the symbols that represent arithmetic math operations. Examples include + (addition operator), - (subtraction operator), \* (multiplication operator), and / (division operator).

Symbol	Operation	Usage	Explanation
+	addition	x+y	Adds values on either side of the operator
-	subtraction	x-y	Subtracts the right hand operand from the left hand operand
*	multiplication	x*y	Multiplies values on either side of the operator
/	division	x/y	Divides the left hand operand by the right hand operand
%	modulus	x%y	Divides the left hand operand by the right hand operand and returns remainder

## Relational Operators:

These operators are used for comparison. They return either **true** or **false** based on the comparison result. The operator '==' should not be confused with '='. The relational operators are as follows:

Symbol	Operation	Usage	Explanation
==	equal	x == y	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	not equal	x != y	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	greater than	x > y	Checks if the value of the left operand is greater than the value of the right operand, if yes then condition becomes true
<	less than	x < y	Checks if the value of the left operand is less than the value of the right operand, if yes then condition becomes true.
>=	greater than or equal	x >= y	Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes then condition becomes true.
<=	less than or equal	x <= y	Checks if the value of the left operand is less than or equal to the value of the right operand, if yes then condition becomes true.

## Logical Operators:

These operators take Boolean (True or False | 1 or 0 ) values as input and return Boolean values as output.

Note: In C,C++& Python any non-zero number is treated as true and 0 as false but this doesn't hold for Java.

Symbol	Operation	Usage	Explanation
--------	-----------	-------	-------------

Symbol	Operation	Usage	Explanation
&&	logical AND	x && y	Returns true if both x and y are true else returns false.
	logical OR	x    y	Returns false if neither x nor y is true else returns true
!	logical NOT	! x	Unary operator. Returns true if x is false else returns false.

### Assignment Operators:

Symbol	Operation	Usage	Equivalence	Explanation
=	assignment	x = y		Assigns value from the right side operand(s) to the left side operand.
+=	add assignment	x += y	x = x+y	Adds the right side operand to the left side operand and assigns the result to the left side operand.
-=	subtract assignment	x -= y	x = x-y	Subtracts the right side operand from the left side operand and assigns the result to the left side operand.
*=	multiply assignment	x *= y	x = x*y	Multiplies the right side operand with the left side operand and assigns the result to the left side operand.
/=	divide assignment	x /= y	x = x/y	Divides the left side operand with the right side operand and assigns the result to the left side operand.
%=	modulus assignment	x%=y	x = x%y	Takes modulus using the two operands and assigns the result to the left side operand.

<<=	left shift and assignment	x<<=y	x= x<<y	Shifts the value of x by y bits towards the left and stores the result back in x.
>>=	right shift and assignment	x>>=y	x= x>>y	Shifts the value of x by y bits towards the right and stores the result back in x.
&=	bitwise AND and assignment	x&=y	x= x&y	Does x&y and stores result back in x.
=	bitwise OR and assignment	x =y	x= x y	Does x y and stores result back in x
^=	bitwise XOR and assignment	x^=y	x= x^y	Does x^y and stores result back in x.

## Increment/Decrement Operators:

These are **unary** operators. Unary operators are the operators which require only one operand.

Symbol	Operation	Usage	Explanation
++	Postincrement	x++	Increment x by 1 after using its value
--	Postdecrement	x--	Decrement x by 1 after using its value
++	Preincrement	++x	Increment x by 1 before using its value
--	Predecrement	--x	Decrement x by 1 before using its value

Examples:

Let, x=10 then, after **y=x++**; y=10 and x=11, this is because x is assigned to y before its increment. but if we had written **y=++x**; y=11 and x=11, because x is assigned to y after its increment. Same holds for decrement operators.

## Bitwise Operators:

These operators are very useful and we have some tricks based on these operators. These operators convert the given integers into binary and then perform the required operation, and give back the result in decimal representation.

Symbols	Operation	Usage	Explanation
&	bitwise AND	$x \& y$	Sets the bit to the result if it is set in both operands.
	bitwise OR	$x   y$	Sets the bit to the result if it is set in either operand.
^	bitwise XOR	$x \wedge y$	Sets the bit if it is set in one operand but not both
~	bitwise NOT	$\sim x$	Unary operator and has the effect of 'flipping' bits,i.e, flips 1 to 0 and 0 to 1.
<<	left shift	$x \ll y$	The left operand's value is moved left by the number of bits specified by the right operand. It is equivalent to multiplying x by $2^y$
>>	right shift	$x \gg y$	The left operand's value is moved right by the number of bits specified by the right operand.It is equivalent to dividing x by $2^y$

### Examples:

Assume  $x=42$ ,  $y=27$

$x=00101010$

$y=00011011$

$x \& y = 0000\ 1010 = 10$  (in decimal)

$x | y = 0011\ 1011 = 59$

$x \wedge y = 0011\ 0001 = 49$

$\sim x = 1101\ 0101$

$x \ll 2 = 1010\ 1000 = 168$ . Notice, the bits are shifted 2 units to the left and the new bits are filled by 0s.

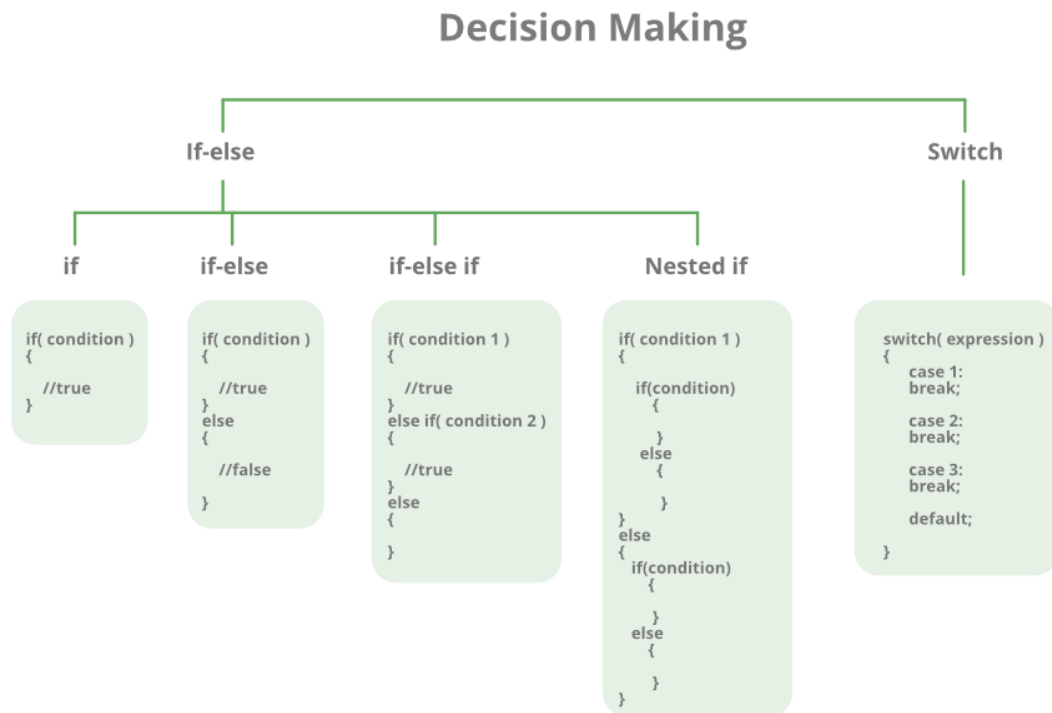
$x \gg 2 = 0000\ 1010 = 10$ \$. Notice, the bits are shifted 2 units to the right and the new bits are filled by 0s.

For more information about how these

operators work, see : [Bit Manipulation](#)

## 7. Decision Making

There come situations in real life when we need to make some decisions and based on those decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on those decisions, we will execute the next block of code. For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute



r. This condition of C else-if is one of the many ways of importing multiple conditions.

1. if statement
2. if..else statements
3. nested if statements
4. if-else-if ladder
5. switch statements
6. Jump Statements:
  - break
  - continue
  - goto
  - return

## 1. if statement in C

If statement is the simplest decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e. if a certain condition is true then a block of statement is executed otherwise not.

```
if(condition)
{
    // Statements to execute if condition is true
}
```

```
if(condition)
    // Statements to execute if condition is true
```

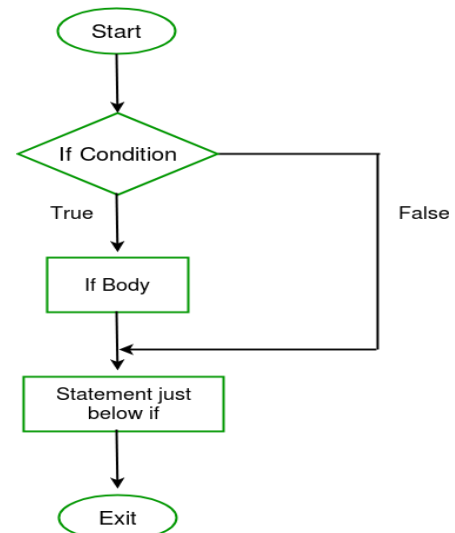
Syntax:

Here, condition after evaluation will be either true or false. C if statement accepts Boolean values (True or False || 1 or 0)– if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Example:

```
if(condition)
    statement1;
    statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```



Example in C:

```
#include <stdio.h>

void main() {
    int i = 10;

    if (i > 15)
    {
        printf("10 is less than 15");
    }

    printf("I am Not in if");
}
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
I am Not in if
PS C:\Users\harsh\Desktop\Practice>
```

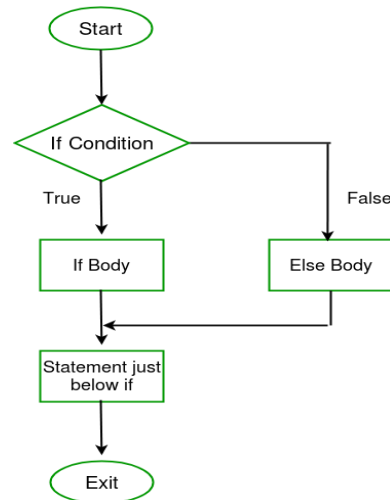
Output

## 2. if-else in C

The 'if' statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```



// C program to illustrate If statement

```
#include <stdio.h>

void main() {
    int i = 20;
    if (i < 15) // for 1 liner we can skip using {}
        printf("i is smaller than 15");
    else
        printf("i is greater than 15");
}
```

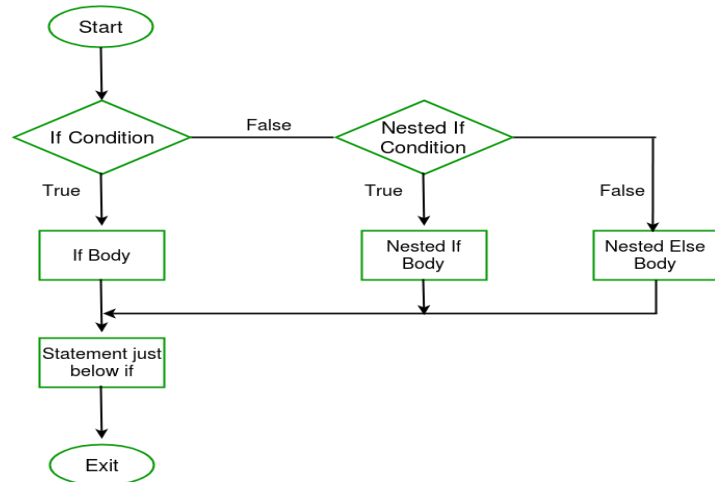


### 3. nested-if in C

A nested if in C is an **if** statement that is the target of another if statement. Nested if statement means an **if** statement inside another if statement. Yes, both C and C++ allows us to nested if statements within if statements, i.e., we can place an **if** statement inside another if statement.

Syntax:

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}
```



```
// C program to illustrate nested-if statement
#include <stdio.h>
void main() {
    int i = 10;
    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");
        // Nested - if statement
        // Will only be executed if statement above is true
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }
}
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
i is smaller than 15
i is smaller than 12 too
PS C:\Users\harsh\Desktop\Practice>
```

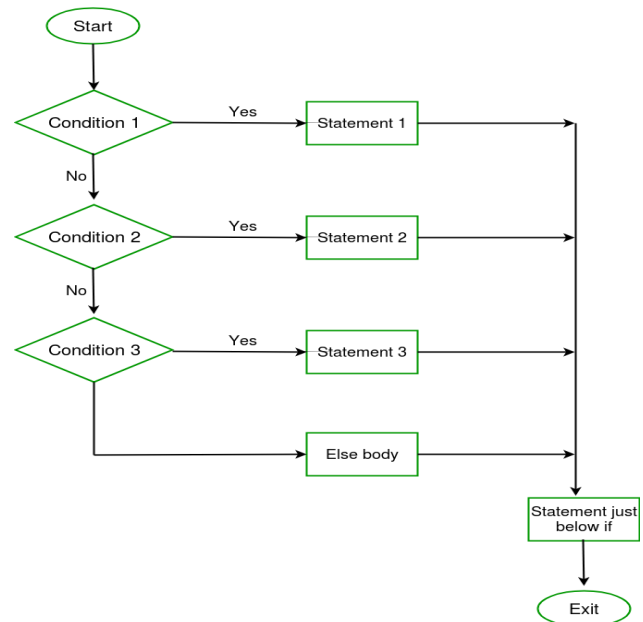
### 4. if-else-if ladder in C

Here, a user can decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

Syntax:

Author: `printf("Electroverts Lab");`

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```



```
// C program to illustrate nested-if statement
#include <stdio.h>
```

```
void main() {
    int i = 20;

    if (i == 10)
        printf("i is 10");
    else if (i == 15)
        printf("i is 15");
    else if (i == 20)
        printf("i is 20");
    else
        printf("i is not present");
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
i is 20
PS C:\Users\harsh\Desktop\Practice>
```

## 5. Jump Statements in C

(Read this after understanding [Loop in next topic](#))

These statements are used in C or C++ for unconditional flow of control throughout the function in a program. They support four type of jump statements:

### 1. C break:

The break statement in C programming has the following two usages –

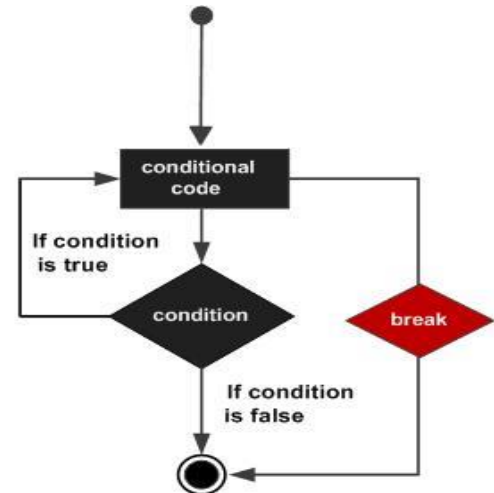
1. When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement ([covered in the next chapter](#)). If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

break;

```
#include <stdio.h>

void main () {
/* local variable definition */
    int a = 10;
/* while loop execution */
    while( a < 20 ) {
        printf("value of a: %d\n", a);
        a++;
        if( a > 15) {
/* terminate the loop using break statement */
            break;
        }
    }
}
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
PS C:\Users\harsh\Desktop\Practice>
```

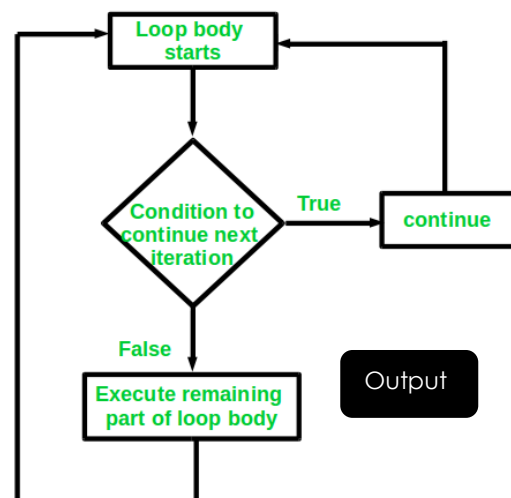
## 2. C continue:

This loop control statement is just like the break statement. The continue statement is opposite to that of break statement, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggest the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and next iteration of the loop will begin.

Syntax:

continue;



```
// C program to explain the use
// of continue statement
#include <stdio.h>

void main() {
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            printf("%d ", i); } }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\hars
1 2 3 4 5 7 8 9 10
PS C:\Users\harsh\Desktop\Practice> |
```

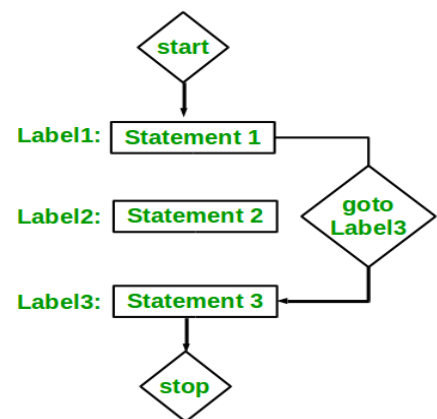
### 3. C goto:

The goto statement in C/C++ also referred to as unconditional jump statement can be used to jump from one point to another within a function.

Syntax:

goto label;	label:
.	.
.	.
label:	goto label;

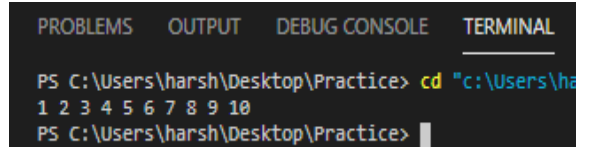
In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier which indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



```
// C program to print numbers
// from 1 to 10 using goto statement
#include <stdio.h>

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ",n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
void main() {
    printNumbers();
}
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
1 2 3 4 5 6 7 8 9 10
PS C:\Users\harsh\Desktop\Practice>
```

#### 4. C return: ([Will understand it better in Functions topic](#))

The return in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and return the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value is must be returned.

Syntax:

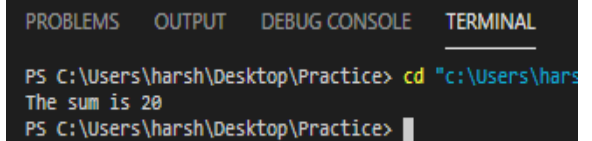
return[expression];

```
// C code to illustrate return
// statement
#include <stdio.h>

// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print
void Print(int s2)
{
    printf("The sum is %d", s2);
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0; }
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
The sum is 20
PS C:\Users\harsh\Desktop\Practice>
```

## 6. Switch Statement

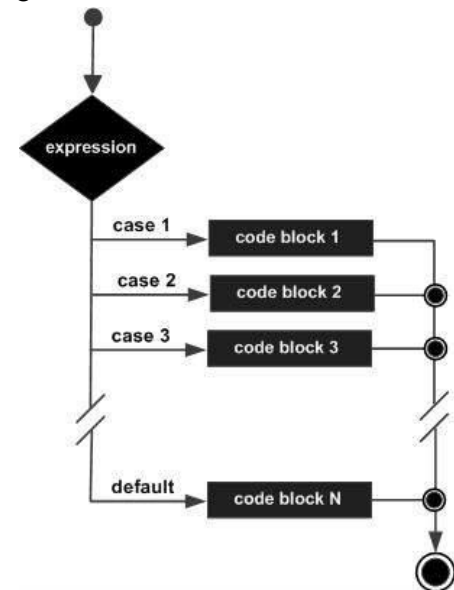
A switch statement is an alternative of if statements which allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case. It has the following syntax –

```
switch(expression){  
    case ONE :  
        statement(s);  
        break;  
    case TWO:  
        statement(s);  
        break;  
    .....  
  
    default :  
        statement(s);  
}
```

The expression used in a switch statement must give an integer value, which will be compared for equality with different cases given. Wherever an expression value matches with a case value, the body of that case will be executed and finally, the switch will be terminated using a break statement. If no break statements are provided, then the computer continues executing other statements available below to the matched case. If none of the cases matches, then the default case body is executed.

The above syntax can be represented in the form of a flow diagram as shown below –

```
#include <stdio.h>  
  
int main() {  
    int x = 2;  
  
    switch( x ){  
        case 1 :  
            printf( "One\n");  
            break;  
        case 2 :  
            printf( "Two\n");  
            break;  
        case 3 :  
            printf( "Three\n");  
            break;  
        case 4 :  
            printf( "Four\n");  
            break;  
        default :  
            printf( "None of the above...\n");  
    }  
}
```



[Author: printf\("Electroverts Lab"\);](#)

When the above program is executed, it produces the following result -

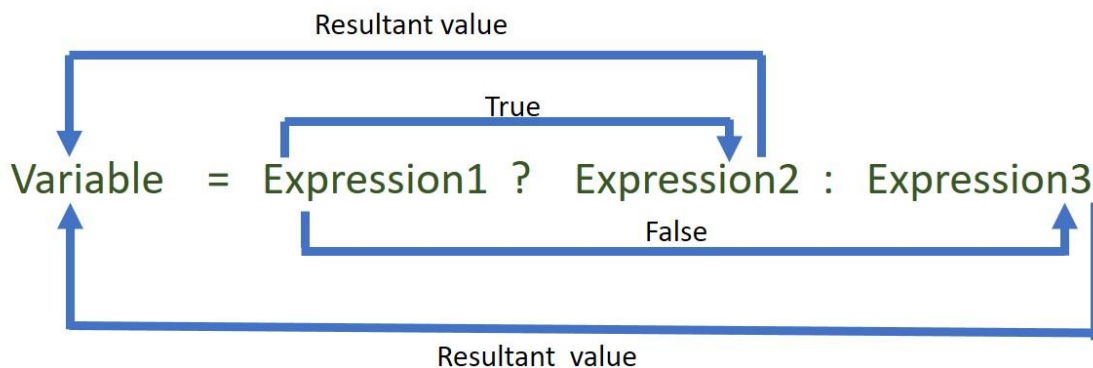
Two

## 7. Ternary Operator

Let's first understand what is ternary/conditional operator.

The conditional operator is kind of similar to the if-else statement as it does follow the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

### Conditional or Ternary Operator (?:) in C++/C



Syntax: **variable = Expression1 ? Expression2 : Expression3**

It can be visualized into if-else statement as:

```
if(Expression1)
{
    variable = Expression2;
}
else
{
    variable = Expression3;
}
```

Since the Conditional Operator '?:' takes three operands to work, hence they are also called ternary operators.

Working:

[Author: printf\("Electroverts Lab"\);](#)

Here, **Expression1** is the condition to be evaluated. If the condition(**Expression1**) is True then **Expression2** will be executed and the result will be returned. Otherwise, if the condition(**Expression1**) is false then **Expression3** will be executed and the result will be returned.

```
// C program to find largest among two
// numbers using ternary operator

#include <stdio.h>

int main()
{
    // variable declaration
    int n1 = 5, n2 = 10, max;

    // Largest among n1 and n2
    max = (n1 > n2) ? n1 : n2;

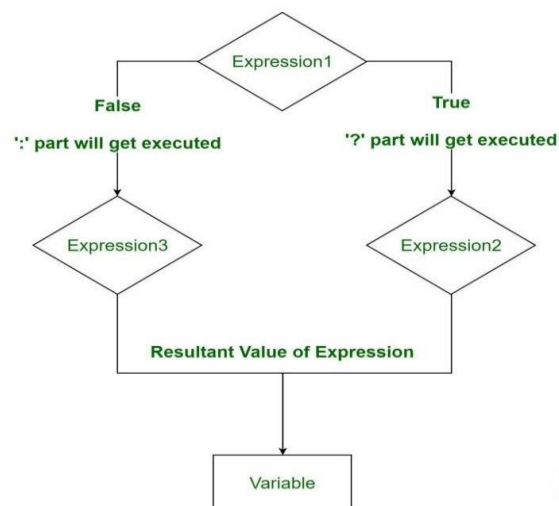
    // Print the largest number
    printf("Largest number between"
        " %d and %d is %d. ",
        n1, n2, max);

    return 0;
}
```

Output:

Largest number between 5 and 10 is 10.

#### Flow Chart of Conditional or Ternary Operator

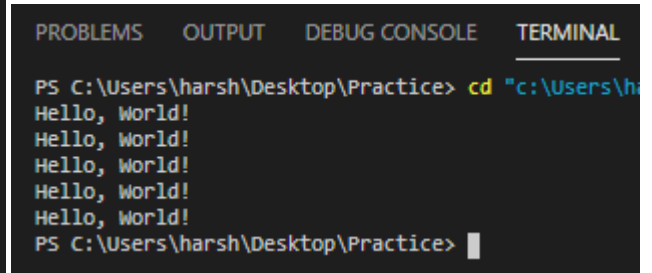




## 8. Loops

Let's consider a situation when you want to print Hello, World! five times. Here is a simple C program to do the same –

```
1  #include <stdio.h>
2
3  int main() {
4      printf( "Hello, World!\n");
5      printf( "Hello, World!\n");
6      printf( "Hello, World!\n");
7      printf( "Hello, World!\n");
8      printf( "Hello, World!\n");
9  }
```

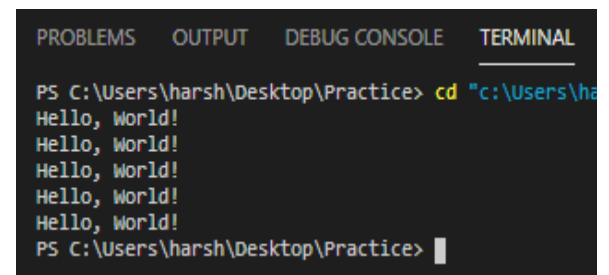


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
PS C:\Users\harsh\Desktop\Practice>
```

It was simple, but again, let's consider another situation when you want to write Hello, World! a thousand times. We can certainly not write `printf()` statements a thousand times. Almost all the programming languages provide a concept called loop, which helps in executing one or more statements up to a desired number of times. All high-level programming languages provide various forms of loops, which can be used to execute one or more statements repeatedly.

Let's write the above C program with the help of a while loop and later, we will discuss how this loop works:

```
1  #include <stdio.h>
2
3  int main() {
4      int i = 0;
5
6      while ( i < 5 ) {
7          printf( "Hello, World!\n");
8          i = i + 1;
9      }
10 }
```



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
PS C:\Users\harsh\Desktop\Practice>
```

The above program makes use of a while loop, which is being used to execute a set of programming statements enclosed within {...}. Here, the computer first checks whether the given condition, i.e., variable "i" is less than 5 or not and if it finds the condition is true, then the loop body is entered to execute the given statements. Here, we have the following two statements in the loop body –

First statement is **printf()** function, which prints Hello World!

[Author: printf\("Electroverts Lab"\);](#)

Second statement is  $i = i + 1$ , which is used to increase the value of variable  $i$ .

After executing all the statements given in the loop body, the computer goes back to `while( i < 5)` and the given condition,  $(i < 5)$ , is checked again, and the loop is executed again if the condition holds true. This process repeats till the given condition remains true which means variable "a" has a value less than 5.

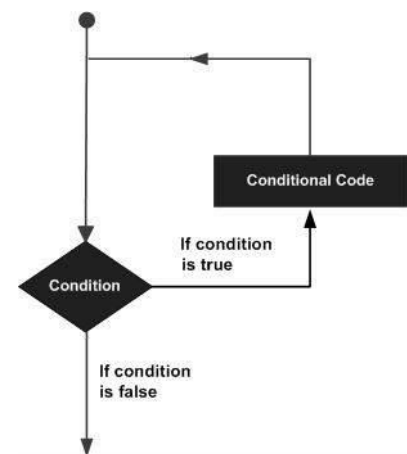
To conclude, a loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages -

## 1. while loop in C

While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean (True or False) condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax:

```
while(condition){  
    //code to be executed  
}
```



Output

```
1  #include<stdio.h>  
2  int main(){  
3  int i=1;  
4  while(i<=10){  
5  printf("%d \n",i);  
6  i++;  
7  }  
8  return 0;  
9  }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
PS C:\Users\harsh\Desktop\Practice>
```

## 2. for Loop

The syntax of the **for** loop is:

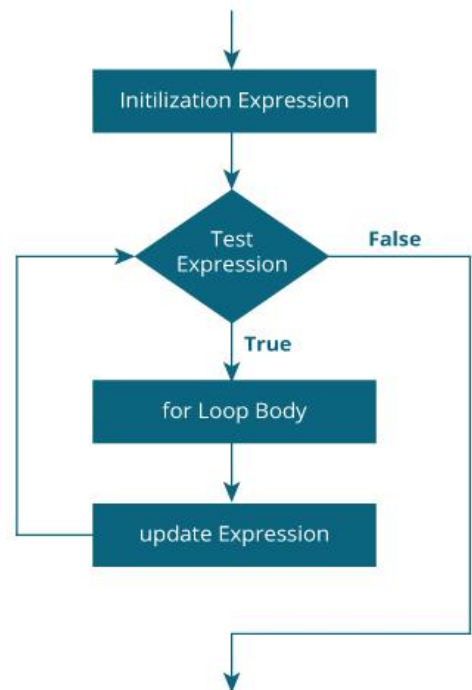
```
for (initializationStatement; testExpression; updateStatement)  
{
```

[Author: printf\("Electroverts Lab"\);](#)

```
// statements inside the body of loop  
}
```

#### Here is the flow of control in a 'for' loop –

- The [initializationStatement](#) step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the [testExpression](#) is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the [updateStatement](#). This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.



#### Example 1: for loop

```
// Print numbers from 1 to 10  
#include <stdio.h>  
int main() {  
    int i;  
  
    for (i = 1; i < 11; i++) // i = 1 is initializationStatement  
    // i < 11 is testExpression which will be tested before e  
    // ++i is update statement which will update i from 1 to  
    // we can put update statement as ++i or i++ as both will  
    // increase i by 1  
    {  
        printf("%d \n", i);  
    }  
    return 0;  
}
```

What will be the output if we remove `\n` from the `printf` statement?

[Author: printf\("Electroverts Lab"\);](#)

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
10
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\..."
1
2
3
4
5
6
7
8
9
10
PS C:\Users\harsh\Desktop\Practice> []
```

Output

### Example 1 explanation

1.  $i$  is initialized to 1.
2. The test expression  $i < 11$  is evaluated. Since 1 less than 11 is true, the body of for loop is executed. This will print the 1 (value of  $i$ ) on the screen.
3. The update statement  $++i$  is executed. Now, the value of  $i$  will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print 2 (value of  $i$ ) on the screen.
4. Again, the update statement  $++i$  is executed and the test expression  $i < 11$  is evaluated. This process goes on until  $i$  becomes 11.
5. When  $i$  becomes 11,  $i < 11$  will be false, and the for loop terminates.

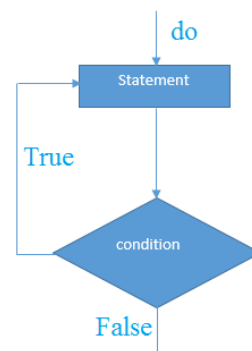
## 3. [do while loop in C](#)

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

do while loop syntax

The syntax of the C language do-while loop is given below:

```
do{
//code to be executed
}while(condition);
```



```
#include<stdio.h>
int main(){
int i=1;
do{
```

[Author: printf\("Electroverts Lab"\);](#)

```
printf("%d \n",i);  
i++;  
}while(i<=10);  
return 0;  
}
```

Output:

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

[PRINTF is discussed here](#), there is another function that comes under [stdio.h library](#).

## [scanf\(\)](#)

The scanf function allows us to accept input from standard input device, which for us is generally the keyboard. The scanf function can do a lot of different things, but can be unreliable because it doesn't handle human errors very well. But for simple programs it's good enough and easy to use.

The simplest application of scanf looks like this:

```
scanf("%d", &b); // here b is variable name where we will store the input and %d signifies that the  
// input datatype will be int.
```

The program will read in an integer value that the user enters on the keyboard (%d is for integers, as is printf, so b must be declared as an int) and place that value into b named variable.

The scanf function uses the same placeholders as printf:

**int** uses %d

**float** uses %f

**char** uses %c

Refer page number 70 for format specifier/place holders.

We must put & in front of the variable used in scanf. "&" before a variable name means "use the address of this variable". So in the scanf function shown above "%d" (integer value will be taken as input and stored in "&b" a variable whose name is b.

The printf and scanf functions will take a bit of practice to be completely understood, but once mastered they are extremely useful.

## Example 2: for loop

```
1 // Program to calculate the sum of first n natural numbers
2 // Positive integers 1,2,3...n are known as natural numbers
3 #include <stdio.h>
4 int main()
5 {
6     int num, count, sum = 0;
7     printf("Enter a positive integer: ");
8     scanf("%d", &num); // taking the integer type input
9     // and storing it in num variable that we created above
10    // for loop terminates when num is less than count
11    for(count = 1; count <= num; ++count)
12    {
13        sum += count; // we can also write
14        // sum = sum + count;
15    }
16
17    printf("Sum = %d", sum);
18
19    return 0;
20 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter a positive integer: 
```

// program is asking for an input

//we  
gave  
input as  
12

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter a positive integer: 12
```

// our program should now give us  
sum of first 12 natural numbers.

### Final Output

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter a positive integer: 12
Sum = 78
PS C:\Users\harsh\Desktop\Practice> 
```

Okay, we can calculate above sum of n natural numbers by formula too.

Sum of n natural numbers =  $n(n+1)/2$ .

Let's code:

```
1 // Program to calculate the sum of first n natural numbers
2 // Positive integers 1,2,3...n are known as natural numbers
3 #include <stdio.h>
4 int main()
5 {
6     int num, sum;
7     printf("Enter a positive integer: ");
8     scanf("%d", &num); // taking the integer type input
9     // and storing it in num variable that we created above
10
11    // now applying formula of n(n+1)/2
12    sum = num * (num + 1) / 2;
13
14    printf("Sum = %d", sum);
15
16    return 0;
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter a positive integer: 12
Sum = 78
PS C:\Users\harsh\Desktop\Practice> 
```

### Example 2 explanation

The value entered by the user is stored in the variable num. Suppose, the user entered 10.

The count is initialized to 1 and the test expression is evaluated. Since the test expression `count<=num` (1 less than or equal to 10) is true, the body of for loop is executed and the value of sum will equal to 1.

Then, the update statement `++count` is executed and the count will equal to 2. Again, the test expression is evaluated. Since 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the sum will equal 3.

This process goes on and the sum is calculated until the count reaches 11.

When the count is 11, the test expression is evaluated to 0 (false), and the loop terminates.

Then, the value of sum is printed on the screen.

If we want to take more than 1 input at a time, our syntax will be

**`scanf("%d %d %d", &variable1, &variable2, &variable3);`**

## 9. Numbers

Every programming language provides support for manipulating different types of numbers such as simple whole integers and floating point numbers. C, Java, and Python categorize these numbers in several categories based on their nature.

Let's go back and check the data type chapter, where we listed down the core data types related to numbers –

Type	Keyword	Value range which can be represented by this data type
Number	int	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
Small Number	short	-32,768 to 32,767
Long Number	long	-2,147,483,648 to 2,147,483,647
Decimal Number	float	1.2E-38 to 3.4E+38 till 6 decimal places

These data types are called primitive data types and you can use these data types to build more data types, which are called user-defined data types.

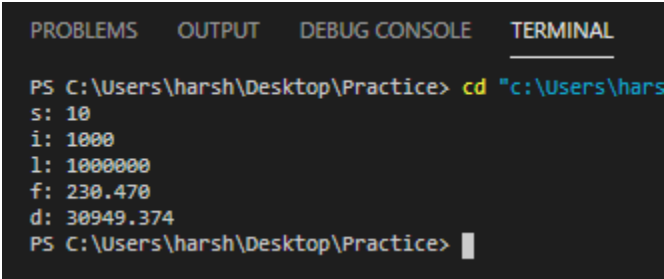
We have seen various mathematical and logical operations on numbers during a discussion on operators. So we know how to add numbers, subtract numbers, divide numbers, etc.

First let's see how to print various types of numbers available in C programming language.

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

```
int main() {  
    short s;  
    int i;  
    long l;  
    float f;  
    double d;  
  
    s = 10;  
    i = 1000;  
    l = 1000000;  
    f = 230.47;  
    d = 30949.374;  
  
    printf("s: %d\n", s);  
    printf("i: %d\n", i);  
    printf("l: %ld\n", l);  
    printf("f: %.3f\n", f);  
    printf("d: %.3f\n", d);  
}
```

Output



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL  
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\hars  
s: 10  
i: 1000  
l: 1000000  
f: 230.470  
d: 30949.374  
PS C:\Users\harsh\Desktop\Practice> |
```



[Author: printf\("Electroverts Lab"\);](#)

Rest of the coding is very obvious, but we used %.3f (or %0.3f) to print float and double, which indicates the number of digits after the decimal to be printed. When the above program is executed, it produces the following result –

```
s: 10
i: 1000
l: 1000000
f: 230.470
d: 30949.374
```

## Math Operations on Numbers:

The following table lists down various useful built-in mathematical functions available in C programming language which can be used for various important mathematical calculations.

For example, if you want to calculate the square root of a number, for example, 2304, then you have a built-in function available to calculate the square root.

Sr.No.	Function & Purpose
1	double <b>cos</b> (double); // double is datatype here This function takes an angle (as a double) and returns the cosine.
2	double <b>sin</b> (double); This function takes an angle (as a double) and returns the sine.
3	double <b>tan</b> (double); This function takes an angle (as a double) and returns the tangent.
4	double <b>log</b> (double); This function takes a number and returns the natural log of that number.
5	double <b>pow</b> (double, double); The first is a number you wish to raise and the second is the power you wish to raise it to.
6	double <b>hypot</b> (double, double); If you pass this function the length of two sides of a right triangle, it will return the length of the hypotenuse.
7	double <b>sqrt</b> (double); You pass this function a number and it returns its square root.
8	int <b>abs</b> (int); This function returns the absolute value of an integer that is passed to it.

[Author: printf\("Electroverts Lab"\);](#)

9	<code>double fabs(double);</code> This function returns the absolute value of any decimal number passed to it.
10	<code>double floor(double);</code> Finds the integer which is less than or equal to the argument passed to it.

Following is a simple example to show a few mathematical operations. To utilize these functions, you need to include the math header file <math.h> in your program in the same way we included stdio.h –

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main() {
5      short s = 10;
6      int i = 1000;
7      long l = 1000000;
8      float f = 230.47;
9      double d = 30.3567;
10
11     printf( "sin(s): %f\n", sin(s));
12     printf( "cos(i): %f\n", cos(i));
13     printf( "floor(f): %f\n", floor(f));
14     printf( "sqrt(l): %f\n", sqrt(l));
15     printf( "pow(d, 2): %f\n", pow(d, 2));
16 }
```

Output

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\
sin(s): -0.544021
cos(i): 0.562379
floor(f): 230.000000
sqrt(l): 1000.000000
pow(d, 2): 921.529235
PS C:\Users\harsh\Desktop\Practice> |

## 10. Characters

If it was easy to work with numbers in computer programming, it would be even easier to work with characters. Characters are simple alphabets like a, b, c, d....., A, B, C, D....., but with an exception. In computer programming, any single digit number like 0, 1, 2.....and special characters like \$, %, +, -.... etc., are also treated as characters and to assign them in a character type variable, you simply need to put them inside single quotes. For example, the following statement defines a character type variable `ch` and we assign a value 'a' to it –

```
char ch = 'a';
```

Here, **ch** is a variable of character type which can hold a character of the implementation's character set and **'a'** is called a **character literal** or a character constant. Not only a, b, c..... but when any number like 1, 2, 3.... or any special character like !, @, #, #, \$,.... is kept inside single quotes, then they will be treated as a character literal and can be assigned to a variable of character type, so the following is a valid statement –

```
char ch = '1';
```

A character data type consumes 8 bits/1 byte of memory which means you can store anything in a character whose ASCII value lies in between -127 to 127, so it can hold any of the 256 different values. A character data type can store any of the characters available on your keyboard including special characters like !, @, #, #, \$, %, ^, &, \*, (, ), \_, +, {, }, etc.

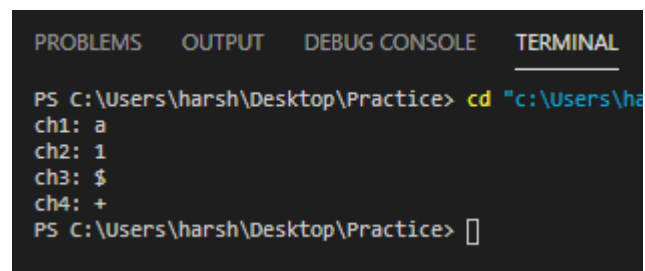
**Note** that you can keep only a single alphabet or a single digit number inside single quotes and more than one alphabets or digits are not allowed inside single quotes. So the following statements are invalid in C programming –

```
char ch1 = 'ab'; // invalid as 1 char stores only 1 alphabet
```

```
char ch2 = '10';
```

Given below is a simple example, which shows how to define, assign, and print characters in C Programming language –

```
1  #include <stdio.h>
2  int main() {
3      char  ch1;
4      char  ch2;
5      char  ch3;
6      char  ch4;
7
8      ch1 = 'a';
9      ch2 = '1';
10     ch3 = '$';
11     ch4 = '+';
12
13     printf( "ch1: %c\n", ch1);
14     printf( "ch2: %c\n", ch2);
15     printf( "ch3: %c\n", ch3);
16     printf( "ch4: %c\n", ch4);
17 }
```



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\ha
ch1: a
ch2: 1
ch3: $
ch4: +
PS C:\Users\harsh\Desktop\Practice> 
```

## Escape Sequences

Many programming languages support a concept called Escape Sequence. When a character is preceded by a backslash (\), it is called an escape sequence and it has a special meaning to the compiler. For example, \n in the following statement is a valid character and it is called a new line character –

```
char ch = '\n';
```

Here, character **n** has been preceded by a backslash (\), it has special meaning which is a new line but keep in mind that backslash (\) has special meaning with a few characters only. The following statement will not convey any meaning in C programming and it will be assumed as an invalid statement –

```
char ch = '\1'; //invalid
```

The following table lists the escape sequences available in C programming language –

Escape Sequence	Description
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.(Set cursor at start of current line)
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\	Inserts a backslash character in the text at this point.

The following example shows how the compiler interprets an escape sequence in a print statement –

```
1  #include <stdio.h>
2
3  int main() {
4      char  ch1;
5      char  ch2;
6      char  ch3;
7      char  ch4;
8
9      ch1 = '\t';
10     ch2 = '\n';
11
12     printf( "Test for tabspace %c and a newline %c will start here", ch1, ch2);
13 }
```

[Author: printf\("Electroverts Lab"\);](#)

Output

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Test for tabspace and a newline
will start here
PS C:\Users\harsh\Desktop\Practice> |
```

### What exactly is \r in the C language?

**\r (Carriage Return):** A carriage return in C programming language is a special code that helps to move the cursor/print head to the beginning of the current line. In the ASCII character set, a carriage return has a decimal value of 13. Therefore, it is not really associated with any specific programming language.

The \r has no inherent meaning for the C language, but terminals (aka console) can react to this character in different ways. The most common way for terminal is that carriage return sets the cursor at the start of the current line.

```
1 #include <stdio.h>
2 void main(){
3     printf("I am amrendra\rBahubali");
4 }
5
```

Output

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Bahubaliendra
PS C:\Users\harsh\Desktop\Practice> |
```

Guess what happened here? First I am amrendra will be printed on screen. I am amrendra is of 13 characters (in string space is also considered as a character), then \r escape sequence will take the cursor to 1<sup>st</sup> position of the line and then Bahubali (8 characters) will be printed again overwriting what was initially there in that line. Now  $13 - 8 = 5$  character will remain as it is which is "endra". So our final output will show Bahubaliendra.

What are Carriage return, linefeed & form feed?

**Carriage return** means to return to the beginning of the current line without advancing downward. The name comes from a printer's carriage, as monitors were rare when the name was coined. This is commonly escaped as \r, abbreviated **CR**.

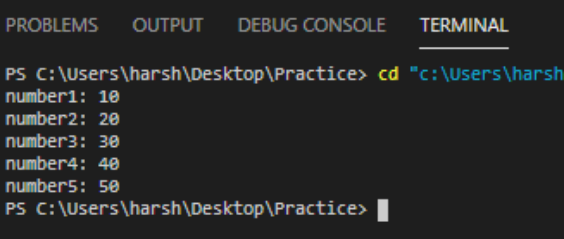
**Linefeed** means to advance downward to the next line; however, it has been repurposed and renamed. Used as "newline", it terminates lines (commonly confused with separating lines). This is commonly escaped as \n, abbreviated **LF** or **NL**.

**Form feed** means advance downward to the next "page". It was commonly used as page separators, but now is also used as section separators. (It's uncommonly used in source code to divide logically independent functions or groups of functions.) Text editors can use this character when you "insert a page break". This is commonly escaped as \f, abbreviated **FF**.

## 11. Arrays

Consider a situation where we need to store five integer numbers. If we use programming's simple variable and data type concepts, then we need five variables of **int** data type and the program will be as follows –

```
1  #include <stdio.h>
2  int main() {
3      int number1 = 10;
4      int number2 = 20;
5      int number3 = 30;
6      int number4 = 40;
7      int number5 = 50;
8
9      printf( "number1: %d\n", number1);
10     printf( "number2: %d\n", number2);
11     printf( "number3: %d\n", number3);
12     printf( "number4: %d\n", number4);
13     printf( "number5: %d\n", number5);
14 }
```



It was simple, because we had to store just five integer numbers. Now let's assume we have to store 5000 integer numbers. Are we going to use 5000 variables?

To handle such situations, almost all the programming languages provide a concept called array. An array is a data structure, which can store a fixed-size collection of elements of the same data type. **An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.**

Instead of declaring individual variables, such as number1, number2, ..., number99, you just declare one array variable number of integer type and use number1[0], number1[1], and ..., number1[99] to represent individual variables. Here, 0, 1, 2, ....99 are index associated with var variable and they are being used to represent individual elements available in the array.

An array in C or C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. **They are used to store similar type of elements as in the data type must be the same for all elements.** They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C or C++ can store derived data types such as the structures, pointers etc. Given below is the picturesque representation of an array.

`Author: printf("Electroverts Lab");`

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

**Array Length = 9**

**First Index = 0**

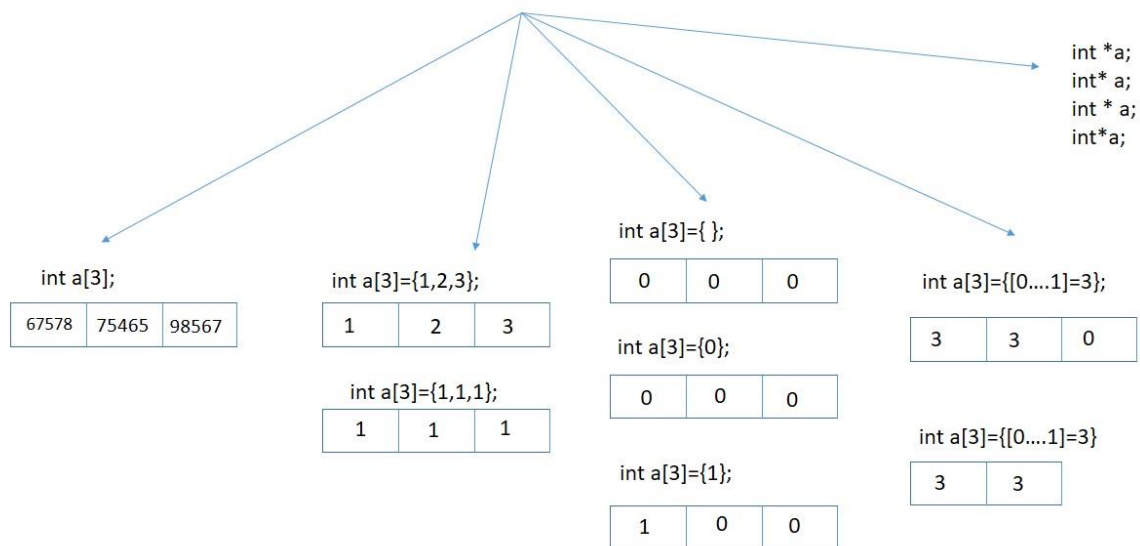
**Last Index = 8**

### Why do we need arrays?

We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

### Array declaration in C/C++:

## ARRAY DECLARATION IN C



There are various ways in which we can declare an array. It can be done by specifying its type and size, by initializing it or both.

### 1. Array declaration by specifying size

```
// Array declaration by specifying size
int arr1[10];
```

// With recent C versions, we can also declare an array of user specified size

```
int n = 10;
```

[Author: printf\("Electroverts Lab"\);](#)

```
int arr2[n];
```

## 2. Array declaration by initializing elements

// Array declaration by initializing elements

```
int arr[] = { 10, 20, 30, 40 }
```

// Compiler creates an array of size 4 and stores given numbers accordingly.

// above is same as "int arr[4] = {10, 20, 30, 40}"

## 3. Array declaration by specifying size and initializing elements

// Array declaration by specifying size and initializing elements

```
int arr[6] = { 10, 20, 30, 40 }
```

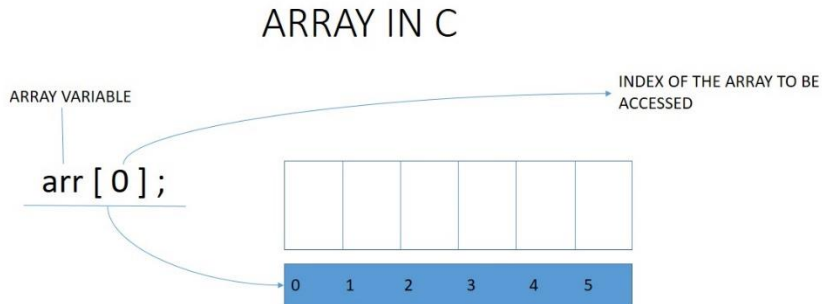
// Compiler creates an array of size 6, initializes first

// 4 elements as specified by user and rest two elements as 0.

// above is same as "int arr[] = {10, 20, 30, 40, 0, 0}"

## Accessing Array Elements:

Array elements are accessed by using an integer index. **Array index starts with 0 and goes till size of array -1.** Let's suppose that we have an array of size 6, indexing will start with 0 and will go on till  $6 - 1 = 5$ . So indexing will be 0,1,2,3,4,5.



```
1 #include <stdio.h>
2 int main()
3 {
4     int arr[5];
5     arr[0] = 5;
6     arr[2] = -10;
7     arr[3 / 2] = 2; // this is same as arr[1] = 2
8     arr[3] = arr[0];
9
10    printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);
11
12    return 0;
13 }
```

PROBLEMS	OUTPUT
PS C:\Users\hars	5 2 -10 5
PS C:\Users\hars	



We can access array elements by their index as seen from example above.

## Some facts on array:

### 1. No Index Out of bound Checking:

There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

```
1 // This C program compiles fine
2 // as index out of bound
3 // is not checked in C.
4 #include <stdio.h>
5 int main()
6 {
7     int arr[2];
8
9     printf("%d ", arr[3]);
10    printf("%d ", arr[-2]);
11
12    return 0;
13 }
14
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"			
3522560 4200704			
PS C:\Users\harsh\Desktop\Practice>			

2. In C, it is not compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just Warning.

```
1 #include <stdio.h>
2 int main()
3 {
4     // Array declaration by initializing it with more
5     // elements than specified size.
6     int arr[4] = { 10, 20, 30, 40, 50 };
7     int lenght = sizeof(arr)/sizeof(arr[0]);
8     for (int i = 0; i < lenght ; i++) {
9         printf("%d \n", arr[i]);
10    }
11    return 0;
12 }
```

Output

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"			
Test.c: In function 'main':			
Test.c:6:33: warning: excess elements in array initializer			
6   int arr[4] = { 10, 20, 30, 40, 50 };			
~~~~~			
Test.c:6:33: note: (near initialization for 'arr')			
10			
20			
30			
40			
PS C:\Users\harsh\Desktop\Practice>			

[Author: printf\("Electroverts Lab"\);](#)

Let's understand above code:

1. We declared array of size 4 but gave 5 values.
2. To get length of array we can use size of function which gives size of a variable in bytes. So, we divided total size of array by size of one element.
3. Then we applied a for loop where initializationStatement = "i" whose value is declared as 0 initially, testExpression <= length of array & updateStatement = increment of 1 in i for each loop.
4. So basically for loop will start from 0 and value of i will keep updating till i becomes greater than length of the array. Here the value of i in for loop will be 0, 1, 2, 3.
5. We keep on printing different values of array by providing them different index position.

### 3. The elements are stored at contiguous memory locations

Example:

```
1 // C program to demonstrate that array elements are stored
2 // at contiguous locations
3 #include <stdio.h>
4 int main()
5 {
6     // an array of 10 integers. If arr[0] is stored at
7     // address x, then arr[1] is stored at x + sizeof(int)
8     // arr[2] is stored at x + sizeof(int) + sizeof(int)
9     // and so on.
10    int arr[5], i;
11
12    printf("Size of integer in this compiler is %lu\n", sizeof(int));
13
14    for (i = 0; i < 5; i++)
15        // The use of '&' before a variable name, yields
16        // address of variable.
17        printf("Address arr[%d] is %p\n", i, &arr[i]);
18
19    return 0;
20 }
```

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Size of integer in this compiler is 4
Address arr[0] is 0061FF08
Address arr[1] is 0061FF0C
Address arr[2] is 0061FF10
Address arr[3] is 0061FF14
Address arr[4] is 0061FF18
PS C:\Users\harsh\Desktop\Practice> |
```

Please read code and comments for understanding how array is storing data and where.

Contiguous (meaning): Sequence

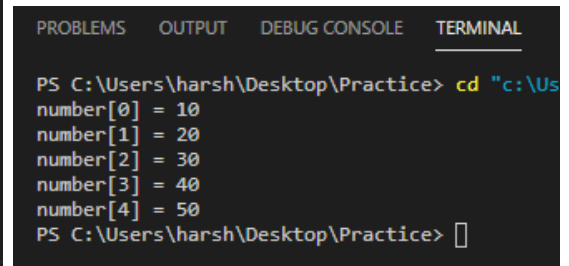
## 12. Strings

During our discussion about **characters**, we learnt that character data type deals with a single character and you can assign any character from your keyboard to a character type variable.

Now, let's move a little bit ahead and consider a situation where we need to store more than one character in a variable. We have seen that C programming does not allow to store more than one character in a character type variable. So the following statements are invalid in C programming and produce syntax errors – `char ch1 = 'ab';`  
`char ch2 = '10';`

We have also seen how to use the concept of **arrays** to store more than one value of similar data type in a variable. Here is the syntax to store and print five numbers in an array of int type –

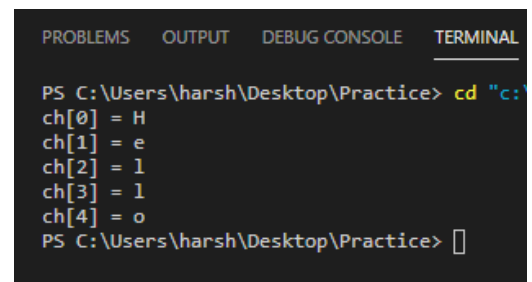
```
1  #include <stdio.h>
2
3  void main() {
4      int number[5] = {10, 20, 30, 40, 50};
5      int i = 0;
6
7      while( i < 5 ) {
8          printf("number[%d] = %d\n", i, number[i] );
9          i = i + 1;
10     }
11 }
```



```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Us
number[0] = 10
number[1] = 20
number[2] = 30
number[3] = 40
number[4] = 50
PS C:\Users\harsh\Desktop\Practice> █
```

Now, let's define an array of five characters in the same way as we did for numbers and try to print them –

```
1  #include <stdio.h>
2
3  void main() {
4      char ch[5] = {'H', 'e', 'l', 'l', 'o'};
5      int i = 0;
6
7      while( i < 5 ) {
8          printf("ch[%d] = %c\n", i, ch[i] );
9          i = i + 1;
10     }
11 }
```



```
PS C:\Users\harsh\Desktop\Practice> cd "c:\
ch[0] = H
ch[1] = e
ch[2] = l
ch[3] = l
ch[4] = o
PS C:\Users\harsh\Desktop\Practice> █
```

Here, we used `%c` to print character value. When the above code is compiled and executed, it produces the following result –

[Author: printf\("Electroverts Lab"\);](#)

If you are done with the above example, then I think you understood how strings work in C programming, because strings in C are represented as arrays of characters. C programming

Simplified the assignment and printing of strings. Let's check the same example once again with a simplified syntax –

```
#include <stdio.h>

void main() {
    char ch[5] = "Hello";
    int i = 0;

    /* Print as a complete string */
    printf("String = %s\n", ch);

    /* Print character by character */
    while( i < 5 ) {
        printf("ch[%d] = %c\n", i, ch[i] );
        i = i + 1;
    }
}
```

Output

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\harsh\Desktop\Practice> cd "C:\Us  
String = Hello  
ch[0] = H  
ch[1] = e  
ch[2] = l  
ch[3] = l  
ch[4] = o  
PS C:\Users\harsh\Desktop\Practice> |

Here, we used %s to print the full string value using array name ch, which is actually the beginning of the memory address holding ch variable as shown below –

Although it's not visible from the above examples, a C program internally assigns null character '\0' as the last character of every string. It indicates the end of the string and it means if you want to store a 5-character string in an array, then you must define an array size of 6 as a good practice, though C does not complain about it.

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Example 1: scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20]; // creating array of 20 digits
    printf("Enter name: ");
    scanf("%s", name); /* taking input as string directly, not by character by character. Also, we can or not use & in scanf because & is used in scanf to store value directly into address but when it comes to character array the pointer variable
```

[Author: printf\("Electroverts Lab"\);](#)

```
itself denotes the base address of the string, thus & is not required. We can use it
though and that will create no problem. */
    printf("Your name is %s.", name); // printing whole string at once
    return 0;
}
```

Output:

Enter name: Electroverts

Your name is Electroverts.

### Basic String Concepts

Based on the above discussion, we can conclude the following important points about strings in C programming language –

- Strings in C are represented as arrays of characters.
- We can constitute a string in C programming by assigning character by character into an array of characters.
- We can constitute a string in C programming by assigning a complete string enclosed in double quote.
- We can print a string character by character using an array subscript or a complete string by using an array name without subscript.
- The last character of every string is a null character, i.e., '\0'.
- Most of the programming languages provide built-in functions to manipulate strings, i.e., you can concatenate strings, you can search from a string, you can extract sub-strings from a string, etc. For more, you can check our detailed tutorial on C programming or any other programming language.

## 13. Functions

A function is a block of statements that performs a specific task. Let's say you are writing a C program and you need to perform a same task in that program more than once. In such case you have two options:

- a) Use the same set of statements every time you want to perform the task.
- b) Create a function to perform that task, and just call it every time you need to perform that task.

Using option (b) is a good practice and a good programmer always uses functions while writing codes in any language.

## Why we need functions in C?

Functions are used because of following reasons –

- a) To improve the readability of code.
- b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
- c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

## Types of functions

### 1) Predefined standard library functions

Standard library functions are also known as built-in functions. Functions such as puts(), gets(), printf(), scanf() etc are standard library functions. These functions are already defined in header files (files with .h extensions are called header files such as stdio.h), so we just call them whenever there is a need to use them.

For example, printf() function is defined in <stdio.h> header file so in order to use the printf() function, we need to include the <stdio.h> header file in our program using #include <stdio.h>.

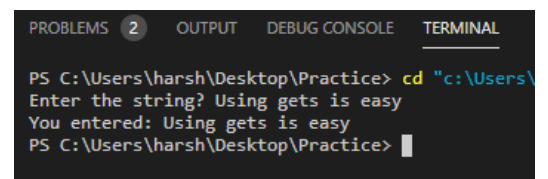
#### gets() and puts() in C programming

The **gets()** function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The **gets()** allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration:

**gets(variable name);** // here we are initializing a character array and storing in any variable and then taking input using gets function and storing that into the array.

```
1  #include<stdio.h>
2  void main ()
3  {
4      char s[30];
5      printf("Enter the string? ");
6      gets(s);
7      printf("You entered: %s",s);
8  }
```



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\
Enter the string? Using gets is easy
You entered: Using gets is easy
PS C:\Users\harsh\Desktop\Practice> |
```

The **gets()** function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered while taking input. It suffers from buffer overflow, which can be avoided by using fgets(). The fgets() makes sure that not more than the maximum limit of characters are read. Consider the following example.

## What is Buffer Overflow?

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the buffer memory. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations.

For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, so if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary.

Declaration of **fgets()**:

`fgets (buffer, size, fp);`

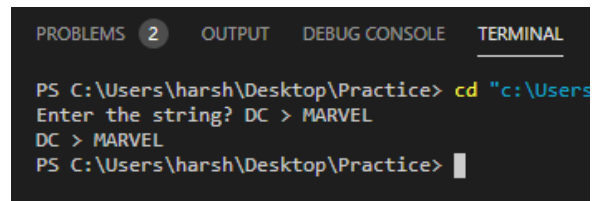
where,

buffer – buffer to put the data in.

size – size of the buffer.

fp – file pointer.

```
1  #include<stdio.h>
2  void main()
3  {
4      char str[20];
5      printf("Enter the string? ");
6      fgets(str, 20, stdin);
7      printf("%s", str);
8  }
```



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter the string? DC > MARVEL
DC > MARVEL
PS C:\Users\harsh\Desktop\Practice>
```

## C puts() function:

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

The image shows a C program snippet on the left and its execution in a terminal window on the right. The code defines a main function that declares a character array 'name' of size 50, prompts the user to enter their name using printf, reads the input using gets, and then displays the string using puts. The terminal output shows the command prompt, the directory change to 'c:\Users\harsh\Desktop\Practice', the prompt 'Enter your name: ', the user input 'Pablo Escobar', and the output 'Pablo Escobar' followed by a newline. Yellow arrows point from the 'puts(name);' line in the code to a box labeled 'puts in action', and from the 'gets(name);' line to a box labeled 'gets in action'.

```
1  ✓ #include<stdio.h>
2  #include <string.h>
3  ✓ void main(){
4      char name[50];
5      printf("Enter your name: ");
6      gets(name); //reads string from user
7      puts(name); //displays string
8  }
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter your name: Pablo Escobar
Pablo Escobar
PS C:\Users\harsh\Desktop\Practice>

puts in action

gets in action

## 2) User Defined functions

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

Now we will learn how to create user defined functions and how to use them in C Programming

### Syntax of a function:

return\_type function\_name (argument list)

{

Set of statements – Block of code

}

**return\_type:** Return type can be of any data type such as int, double, char, void, short etc which the function will return.

**function\_name:** It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.

**argument list:** Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

**Block of code:** Set of C statements, which will be executed whenever a call will be made to the function.



[Author: printf\("Electroverts Lab"\);](#)

Suppose you want to create a function to add two integer variables.

Let's split the problem so that it would be easy to understand –

Function will add the two numbers so it should have some meaningful name like sum, addition, etc. For example let's take the name addition for this function.

return\_type **addition** (argument list)

This function addition adds two integer variables, which means I need two integer variables as input, let's provide two integer parameters in the function signature. The function signature would be –

return\_type **addition(int num1, int num2)**

The result of the sum of two integers would be integer only. Hence function should return an integer value – I got my return type – It would be integer –

**int**addition(int num1, int num2);

So we got your function prototype or signature. Now you can implement the logic in C program like this:

#### **Example1: Creating a user defined function addition()**

```
#include <stdio.h>
int addition(int num1, int num2)
{
    int sum;
    /* Arguments are used here*/
    sum = num1+num2;
    /* Function return type is integer so we are returning
       an integer value, the sum of the passed numbers.*/
    return sum;
}

int main() // now you must have known why we were writing void before main
//function, Void return means we were not expecting any return from the
//main function. Here we used int as return type for main and later in
//program we returned 0.
{
    int var1, var2;
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);

    /* Calling the function here, the function return type
```

[Author: printf\("Electroverts Lab"\);](#)

```
    * is integer so we need an integer variable to hold the
    * returned value of this function.
    */
    int res = addition(var1, var2);
    printf ("Output: %d", res);

    return 0;
}
```

Output

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users
Enter number 1: 10
Enter number 2: 20
Output: 30
PS C:\Users\harsh\Desktop\Practice> █
```

**Example2: Creating a void user defined function that doesn't return anything**

```
#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
    printf("Hi\n");
    printf("My name is Walter White\n");
    printf("How are you?");
    /* There is no return statement inside this function, since its
    return type is void*/
}

int main()
{
    /*calling function*/
    introduction();
    return 0;
}
```

Output

```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users
Hi
My name is Walter White
How are you?
PS C:\Users\harsh\Desktop\Practice> █
```

### Few Points to Note regarding functions in C:

- 1) main() in C program is also a function.
- 2) Each C program must have at least one function, which is main().
- 3) There is no limit on number of functions; A C program can have any number of functions.
- 4) A function can call itself and it is known as "Recursion".

### C Functions Terminologies that you must remember

Return type: Data type of returned value. It can be **void** also, in such case function doesn't return any value.

Note: for example, if function return type is **char**, then function should return a value of **char** type and while calling this function the main() function should have a variable of **char** data type to store the returned value.

Structure would look like –

```
char abc(char ch1, char ch2)
{
    char ch3;
    ...
    ...
    return ch3;
}

int main()
{
    ...
    char c1 = abc('a', 'x');
    ...
}
```

## Additional: Function call by value & call by reference

### Call by Value:

Function call by value is the default way of calling a function in C programming. Before we discuss function call by value, let's understand the terminologies that we will use while explaining this:

**Actual parameters:** The parameters that appear in function calls.

**Formal parameters:** The parameters that appear in function declarations.

**For example:**

```
#include <stdio.h>
int sum(int a, int b)
{
    int c=a+b;
    return c;
}

int main(
{
    int var1 =10;
    int var2 = 20;
    int var3 = sum(var1, var2);
    printf("%d", var3);

    return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

**What is Function Call By value?** When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied to the formal parameters. Thus, operations performed on the formal parameters don't reflect in the actual parameters.

### Example of Function call by Value

As mentioned above, in the call by value the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters. Let's take an example to understand this:

[Author: printf\("Electroverts Lab"\);](#)

```
#include <stdio.h>
int increment(int var)
{
    var = var+1;
    return var;
}

int main()
{
    int num1=20;
    int num2 = increment(num1);
    printf("num1 value is: %d", num1);
    printf("\nnum2 value is: %d", num2);

    return 0;
}
```

Output:

num1 value is: 20

num2 value is: 21

**Explanation:** We passed the variable num1 while calling the method, but since we are calling the function using call by value method, only the value of num1 is copied to the formal parameter var. Thus, change made to the var doesn't reflect in the num1.

### Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
    int tempnum ;
    /*Copying var1 value into temporary variable */
    tempnum = var1 ;
    /* Copying var2 value into var1*/
    var1 = var2 ;
    /*Copying temporary variable value into var2 */
    var2 = tempnum ;
}

int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);
    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output:

Before swapping: 35, 45

[Author: printf\("Electroverts Lab"\);](#)

After swapping: 35, 45

### Why variables remain unchanged even after the swap?

The reason is same – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

### Function call by reference:

Before we discuss function call by reference, let's understand the terminologies that we will use while explaining this:

**Actual parameters:** The parameters that appear in function calls.

**Formal parameters:** The parameters that appear in function declarations.

For example: We have a function declaration like this:

```
int sum(int a, int b);
```

The a and b parameters are formal parameters.

We are calling the function like this:

```
int s = sum(10, 20); //Here 10 and 20 are actual parameters
or
int s = sum(n1, n2); //Here n1 and n2 are actual parameters
```

**What is Function Call by Reference?** When we call a function by passing the addresses of actual parameters then this way of calling the function is known as call by reference. In call by reference, the operation performed on formal parameters, affects the value of actual parameters because all the operations performed on the value stored in the address of actual parameters. It may sound confusing first but the following example would clear your doubts.

### Example of Function call by Reference:

```
#include <stdio.h>
void increment(int *var)
{
    /* Although we are performing the increment on variable
     * var, however the var is a pointer that holds the address
     * of variable num, which means the increment is actually done
     * on the address where value of num is stored.
     */
    *var = *var+1;
}
int main()
{
    int num=20;
    /* This way of calling the function is known as call by
     * reference. Instead of passing the variable num, we are
     * passing the address of variable num
     */
    increment(&num);
    printf("Value of num is: %d", num);
    return 0;
}
```

[Author: printf\("Electroverts Lab"\);](#)

Output:

Value of num is: 21

### Example 2: Function Call by Reference – Swapping numbers

Here we are swapping the numbers using call by reference. As you can see the values of the variables have been changed after calling the swapnum() function because the swap happened on the addresses of the variables num1 and num2.

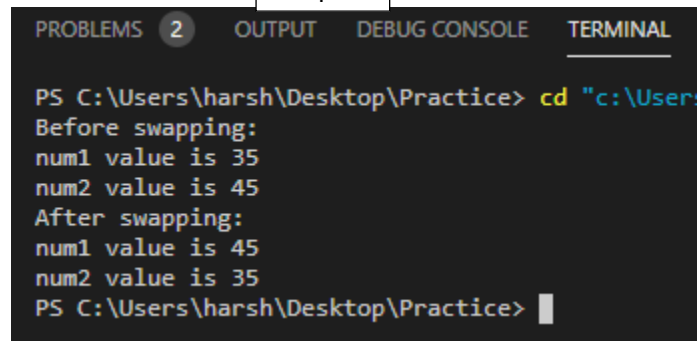
```
#include <stdio.h>
void swapnum ( int *var1, int *var2 )
{
    int tempnum ;
    tempnum = *var1 ;
    *var1 = *var2 ;
    *var2 = tempnum ;
}

int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping:");
    printf("\nnum1 value is %d", num1);
    printf("\nnum2 value is %d", num2);

    /*calling swap function*/
    swapnum( &num1, &num2 );

    printf("\nAfter swapping:");
    printf("\nnum1 value is %d", num1);
    printf("\nnum2 value is %d", num2);
    return 0;
}
```

Output



```
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\
Before swapping:
num1 value is 35
num2 value is 45
After swapping:
num1 value is 45
num2 value is 35
PS C:\Users\harsh\Desktop\Practice> |
```

## 14. Miscellaneous

### 1. floor() and ceil() functions of math.h in C

Some basic mathematical calculations are based on the concept of floor and ceiling. Floor means a whole number which should be less than or equal to the number given and must be nearest to the number.

Ceiling means a whole number which is more than or equal to the value given and also must be nearest to the number. You can look at the example given below for more clarification.

To use floor and ceil functions all you need to do is pass a number as a parameter and these function will return a number satisfying the above-explained concept. You can store the result and use it in whichever way you want to.

#### **math.h - floor() function Example in C**

```
#include <stdio.h>
//to use 'floor()' function
#include <math.h>

int main()
{
    // set the type of variable
    float number, a;

    // message for user
    printf("Please enter a number from keyboard to round it up: ");
    scanf("%f", &number);

    // storing the round up value
    a = floor(number);

    // printing the calculated value
    printf("Calculated round up number is: %.2f\n", a);

    return 0;
}
```

Output:

Please enter a number from keyboard to round it up: 12.45

Calculated round up number is: 12.00

#### **math.h - ceil() function Example in C**

```
#include <stdio.h>
//to use 'ceil()' function
#include <math.h>

int main()
```



[Author: printf\("Electroverts Lab"\);](#)

```
{  
    // set the type of variable  
    float number, a;  
  
    // message for user  
    printf("Please enter a number from keyboard to found out it's ceil value: ");  
    scanf("%f", &number);  
  
    // storing the ceil value  
    a = ceil(number);  
  
    // printing the calculated value  
    printf("Calculated ceil number is: %.3f\n", a);  
  
    return 0;  
}
```

Output:

*Please enter a number from keyboard to found out it's ceil value: 12.12*

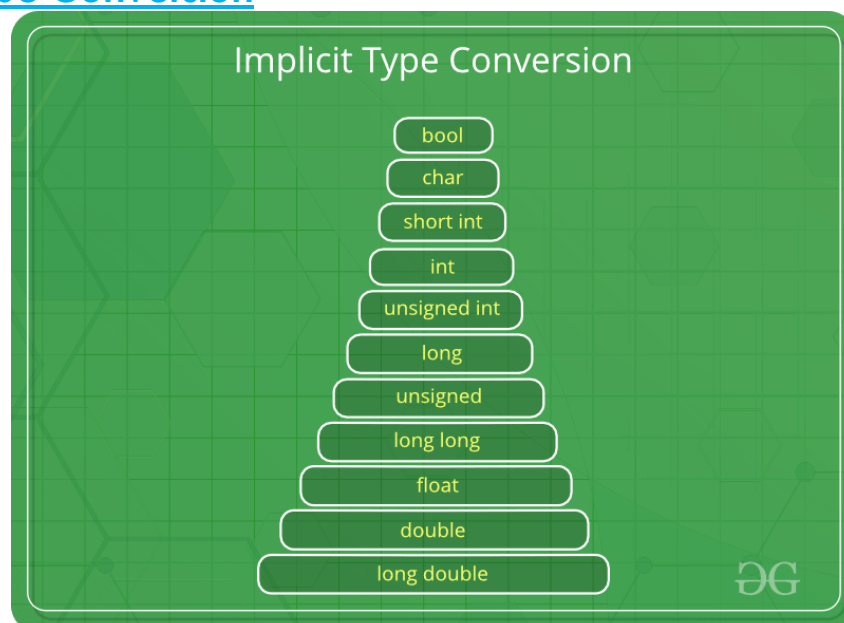
*Calculated ceil number is: 13.000*

## 2. Typecasting in C

Converting one datatype into another is known as type casting or, type-conversion. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

### Implicit Type Conversion



[Author: printf\("Electroverts Lab"\);](#)

Implicit Conversion is also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally, takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

Example of implicit conversion

```
// An example of implicit conversion
#include<stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;

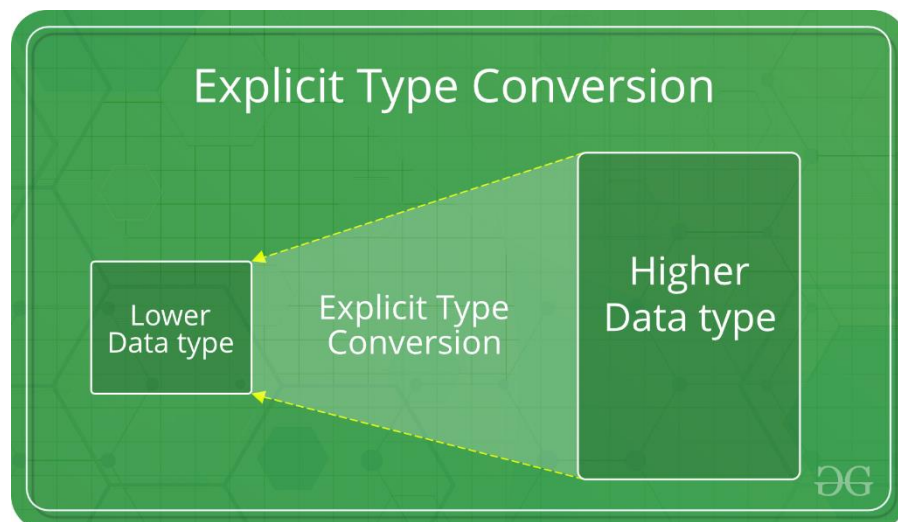
    // x is implicitly converted to float
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}
```

Output:

X = 107, z = 108.000000

## [Explicit Type Conversion](#)



[Author: printf\("Electroverts Lab"\);](#)

This process is also called type casting and it is user defined. Here the user can type cast the result to convert it to a particular data type.

The syntax in C:

(Type) expression

Example of explicit conversion

```
// C program to demonstrate explicit type casting
#include<stdio.h>

int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}
```

Output:

sum = 2

### 3. Prefix and Postfix operators in C

The increment operator ++ increases the value of a variable by 1. Similarly, the decrement operator -- decreases the value of a variable by 1.

For example:

```
a = 5
++a; // a becomes 6
a++; // a becomes 7
--a; // a becomes 6
a--; // a becomes 5
```

- If you use the ++ operator as prefix like: ++var. The value of var is incremented by 1 then, it returns the value.
- If you use the ++ operator as postfix like: var++. The original value of var is returned first then, var is incremented by 1.

Example of usage of prefix and postfix operators

[Author: printf\("Electroverts Lab"\);](#)

```
#include <stdio.h>
int main() {
    int var1 = 5, var2 = 5;

    // var1 is displayed
    // Then, var1 is increased to 6.
    printf("%d\n", var1++);

    // var2 is increased to 6
    // Then, it is displayed.
    printf("%d\n", ++var2);

    return 0;
}
```

Output:

5

6

## 4. Hierarchy of operators and their associativity

- The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %>>= <<= &= ^=  =	Right to left
Comma	,	Left to right

## 6. Scope of Variables in C

- A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration.
- A variable in C can be defined in three places within a program. These are:

Position	Type
Inside a function or a block.	local variables
Out of all functions.	Global variables
In the function parameters.	Formal parameters

### Local Variables

Variables that are declared within the function block and can be used only within the function is called local variables.

### Local Scope

A local scope or block is collective program statements put in and declared within a function or block (a specific region enclosed with curly braces) and variables lying inside such blocks are termed as local variables. All these locally scoped statements are written and enclosed within left ({) and right braces (}) curly braces. There's a provision for nested blocks also in C which means there can be a block or a function within another block or function. So it can be said that variable(s) that are declared within a block can be accessed within that specific block and all other inner blocks of that block, but those variables cannot be accessed outside the block.

## Global Variables

Variables that are declared outside of a function block and can be accessed inside the function is called global variables.

## Global Scope

Global variables are defined outside a function or any specific block, in most of the case, on the top of the C program. These variables hold their values all through the end of the program and are accessible within any of the functions defined in your program.

Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared.

## 7. Headers in C

It is important to understand, what are preprocessor directives? These are the basic building blocks of header files in C. The term "**preprocessor**" is *self-explanatory*. The word "pre" means "before" and the word "processor" means "to make something". Before the source code is compiled, it gets automatically processed due to the presence of preprocessor directives.

A header file in C/C++ contains:

Function definitions

Data type definitions

Macros

Header files offer these features by importing them into your program with the help of a preprocessor directive called `#include`. These preprocessor directives are responsible for instructing the C/C++ compiler that these files need to be processed before compilation.

Every C program should necessarily contain the header file `<stdio.h>` which stands for standard input and output used to take input with the help of `scanf ()` function and display the output using `printf()` function.

## List of commonly used header files in C

Sr.No.	Header Files & Description
1	<b>stdio.h</b> Input/Output functions
2	<b>conio.h</b> Console Input/Output functions
3	<b>stdlib.h</b> General utility functions
4	<b>math.h</b> Mathematics functions
5	<b>string.h</b> String functions
6	<b>ctype.h</b> Character handling functions
7	<b>time.h</b> Date and time functions
8	<b>float.h</b> Limits of float types
9	<b>limits.h</b> Size of basic types
10	<b>wctype.h</b> Functions to determine the type contained in wide character data.

## 8. Macros using #define

A macro is a fragment of code that is given a name. You can define a macro in C using the `#define` preprocessor directive.

Here's an example.

```
#define c 299792458 // speed of light
```

Here, when we use `c` in our program, it is replaced with `299792458`.

## 9. Static Variables in C

Static variables are initialized only once. The compiler persists with the variable till the end of the program. Static variables can be defined inside or outside the function. They are local to the block. The default value of static variables is zero. The static variables are alive till the execution of the program.

Here is the syntax of static variables in C language,

```
static datatype variable_name = value;
```

Here,

datatype – The datatype of variable like int, char, float etc.

variable\_name – This is the name of variable given by user.

value – Any value to initialize the variable. By default, it is zero.

## 9. Two Dimensional Arrays in C

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]



## 10. IDE vs Editor

Well many of us often get confused between a text editor and an IDE. So let's first understand the basic difference between them.

The term "IDE" comes from **I**ntegrated **D**evelopment **E**nvironment. It is intended as a set of tools that all work together: text editor, compiler, build or make integration, debugging, etc.

An editor is simply that, a tool that is designed to edit text.

## 11. <ctype.h>

The <ctype.h> header file declares a set of functions to classify (and transform) individual characters. For example, `isupper()` checks whether a character is uppercase or not.

### C isalnum()

checks alphanumeric character

### C isalpha()

checks whether a character is an alphabet or not

### C isdigit()

checks numeric character

### C isgraph()

checks graphic character

### C islower()

checks lowercase alphabet

### C isprint()

checks printable character

### C ispunct()

checks punctuation

### C isspace()

check white-space character

### C isupper()

checks uppercase alphabet

### C tolower()

converts alphabet to lowercase

## C toupper()

converts to lowercase alphabet

## 12.The Comma Operator

The Comma Operator can either work as a separator or an operator.

For example:

PROGRAM 1

```
#include<stdio.h>

int main(void)
{
    int a;
    a = 1, 2, 3;
    printf("%d", a);
    return 0;
}
```

PROGRAM 2

```
#include<stdio.h>

int main(void)
{
    int a;
    a = (1, 2, 3);
    printf("%d", a);
    return 0;
}
```

Comma works as an operator in PROGRAM 1. Precedence of comma operator is least in operator precedence table. So the assignment operator takes precedence over comma and the expression "a = 1, 2, 3" becomes equivalent to "(a = 1), 2, 3". That is why we get output as 1 in the second program.

In PROGRAM 2, brackets are used so comma operator is executed first and we get the output as 3.

**Enough of theory, now it's time to code. We will keep on coming back to each concept while writing code for different problems.**

# Let's Code

## 1. Installing compiler, IDE & Editor.

We need to have a [Compiler](#) & [Editor/ IDE](#) to start the fun.

Although you can install gcc compiler from any source but I am using MinGW installer.

I am using **gcc** compiler for compiling C, for **windows** users click on following link:

<https://osdn.net/projects/mingw/downloads/68260/mingw-get-setup.exe/>

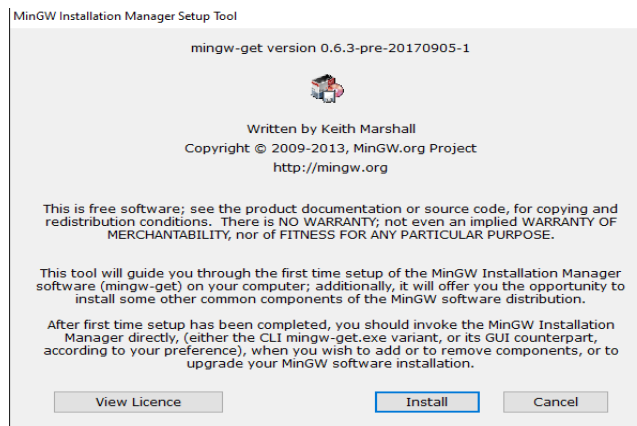
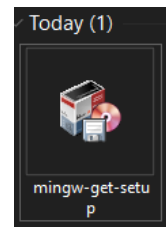
For **ios& linux** users: <https://www.cyberciti.biz/faq/howto-apple-mac-os-x-install-gcc-compiler/>

### Let's get started:

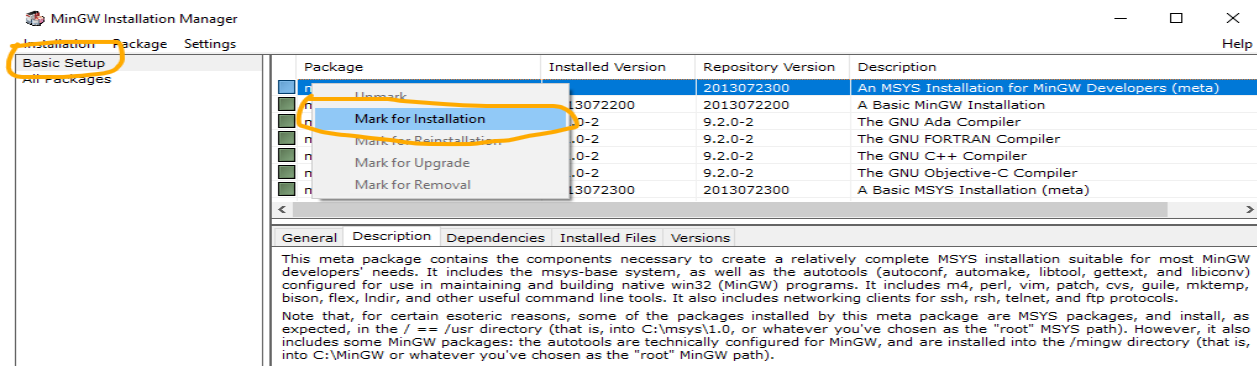
**Step 1:** Click on the link above (ctrl + left-click), following that a page will open in your default browser and download will proceed automatically. Save the file somewhere and run it.

**Step 2:** Now I will run the downloaded software by double clicking on it.

**Step 3:** You already know what to do from here.

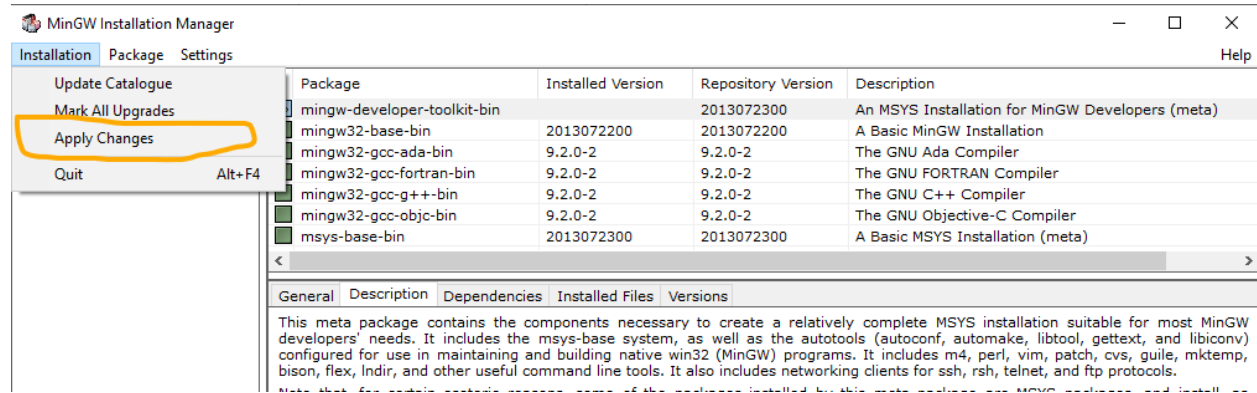


Author: printf("Electroverts Lab");



**Step 4:** After installing wait for MinGW installation manager window to open.

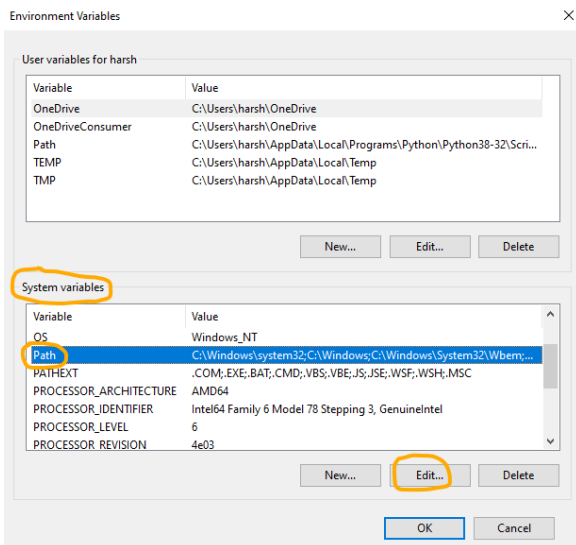
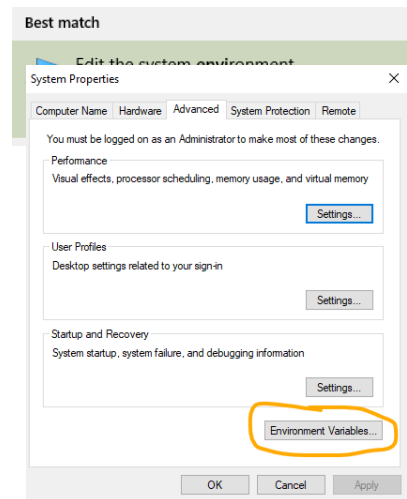
Select basic setup button and right click on each package and click on MARK FOR INSTALLATION.



Click on apply changes and proceed. Installation will be done.

**Step 5:** Setting MinGW in system environment variable path.

Search for "Environment Variables", in search box and click on it. System properties box will open.

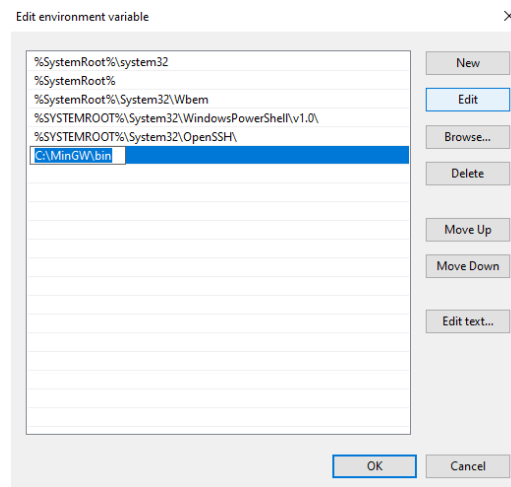


Click on Environment variable

Environment variable dialogue box will open,

[Author: printf\("Electroverts Lab"\);](#)

Then click on New, add MinGW bin folder's path and click OK.



**Note:** We set path because our operating system searches for called binary in path folders only. We downloaded our compiler but we need to tell our operating system that I have stored my compiler at this path and whenever I will call for its use, please search here too and use it if found.

**Step 6:** Till now we downloaded our compiler and set it in path, next step would be to download the [Editor](#).

The term "IDE" comes from Integrated Development Environment. It is intended as a set of tools that all work together: text editor, compiler, build or make integration, debugging, etc. Virtually all IDEs are tied specifically to a language or framework or tightly collected set of languages or frameworks. Some examples: Visual Studio for .NET and other Microsoft languages, RubyMine for Ruby, IntelliJ for Java, XCode for Apple technologies.

An editor is simply that, a tool that is designed to edit text. Typically, they are optimized for programming languages though many programmer's text editors are branching out and adding features for non-programming text like Markdown. The key here is that text editors are designed to work with whatever language or framework you choose.

I use Visual Studio Editor for programming as it is easy and looks great, in my starting days I used Code Blocks IDE, you can use either one of them but VS code Editor is recommended.

How to download? <https://code.visualstudio.com/download>

Download it, install and then open the editor & follow steps told in this video - [https://www.youtube.com/watch?v=77v-Poud\\_io](https://www.youtube.com/watch?v=77v-Poud_io)

**Step 7:** Test it. Print Hello world program in it and check for output.

# Code: Basic Programs

## 1. Hello World Program

Below is a simple program printing Hello World in C language.

[printf\(\)](#) is a system defined function under the header file `stdio.h`, used to print data onto the screen

[\n](#) is used to move the control onto the next line

[\t](#) is used to give a horizontal tab i.e. continuous five spaces

```
#include <stdio.h>

int main()
{
    printf("\n\n\t\tCoding is Fun - Let's learn\n\n\n");
    int num;
    printf("\t\tHello world! Sabka badla lega re tera Faizal\n");
    printf("\n\n\t\tCoding is Fun !\n\n\n");
    return 0;
}
```

### Output

```
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Pr

Coding is Fun - Let's learn

Hello world! Sabka badla lega re tera Faizal

Coding is Fun !
```

## 2. Program to take input of various datatypes in C

Below is a program to explain how to take input from user for different datatypes available in C language. The different [datatypes are int\(integer values\), float\(decimal values\) and char\(character values\)](#).

[printf\(\)](#) is used to display text onto the screen & is used to assign the input value to the variable and store it at that particular location.

[scanf\(\)](#) is used to take input from the user using format specifier, `%d` and `%i`, both are used to take numbers as input from the user. `%f` is the [format specifier](#) to take float as input from the user, `%s` is the format specifier to take character as input from the user.

```
#include<stdio.h>

int main()
{
```

[Author: printf\("Electroverts Lab"\);](#)

```
printf("\n\t\tCoding is Fun - Let's learn\n");

int num1, num2;
float fraction;
char one_char;

printf("Enter two numbers number: ");

// Taking integer as input from user
scanf("%d%i", &num1, &num2); // %i will take decimal, octal, hexadecimal etc as input.
// %d will take only signed decimals as input
printf("\nThe two numbers You have entered are %d and %i\n\n", num1, num2);

// Taking float or fraction as input from the user
printf("Enter a Decimal number: ");
scanf("%f", &fraction);
printf("\nThe float or fraction that you have entered is %0.2f", fraction);
// %0.2f means that only 2 numbers will be printed after demial

// Taking Character as input from the user
printf("\nEnter a Character: ");
scanf("%c",&one_char);
printf("\nThe character that you have entered is %c", one_char);
return 0;
}
```

### 3. C Program to Add Two Integers

In this example, the user is asked to enter two integers. Then, the sum of these two integers is calculated and displayed on the screen.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Data Types](#)
- [C Variables, Constants and Literals](#)
- [C Programming Operators](#)

```
#include <stdio.h>
int main() {

    int number1, number2, sum;

    printf("Enter two integers: ");
```



[Author: printf\("Electroverts Lab"\);](#)

```
scanf("%d %d", &number1, &number2);  
// we are taking integer input that's why %d is used.  
// & {variable name} signifies address of the saved variable.  
  
// calculating sum  
sum = number1 + number2;  
  
printf("%d + %d = %d", number1, number2, sum);  
return 0; // since main function is returning int, we returned 0 here.  
// we could have also wrote void main(){ } and there would be no need to  
// write return statement  
}
```

Output:

Enter two integers: 20

12

20 + 12 = 32

In this program, the user is asked to enter two integers. These two integers are stored in variables number1 and number2 respectively.

**printf("Enter two integers: ");**

**scanf("%d %d", &number1, &number2);**

Then, these two numbers are added using the + operator, and the result is stored in the sum variable.

**sum = number1 + number2;**

Finally, the printf() function is used to display the sum of numbers.

**printf("%d + %d = %d", number1, number2, sum);**

#### **Work to do:**

Write a program to take 3 decimal inputs and add them. (Hint: Use float data type)

## 4. C Program to Multiply Two Floating-Point Numbers

In this example, the product of two floating-point numbers entered by the user is calculated and printed on the screen.

To understand this example, we should have the knowledge of the following C programming topics:

- [C Variables, Constants and Literals](#)
- [C Data Types](#)
- [C Programming Operators](#)

```
#include <stdio.h>
int main() {
    double a, b, product;
    printf("Enter two numbers: ");
    scanf("%lf %lf", &a, &b);

    // Calculating product
    product = a * b;

    // Result up to 2 decimal point is displayed using %.2lf
    printf("Product = %.2lf", product);

    return 0;
}
```

Output:

Enter two numbers: 12.22

13.1

Product = 160.08

In this program, the user is asked to enter two numbers which are stored in variables a and b respectively.

```
printf("Enter two numbers: ");
scanf("%lf %lf", &a, &b);
```

Then, the product of a and b is evaluated and the result is stored in product variable.

```
product = a * b;
```

Finally, product is displayed on the screen using printf().

```
printf("Product = %.2lf", product);
```

Notice that, the result is rounded off to the second decimal place using %.2lf conversion character.

**Float is single precision 32bit floating point data type while double is double precision 64bit floating point data type.**

**Work to do:**

Write a program to multiply 3 floating point numbers using **float** data type.

## 5. C Program to Find ASCII Value of a Character

In C programming, a character variable (Data type char) holds ASCII value (an integer number between 0 and 127) rather than that character itself. This integer value is the ASCII code of the character.

For example, the ASCII value of 'A' is 65.

What this means is that, if we assign 'A' to a character variable, 65 is stored in the variable rather than 'A' itself.

Now, let's see how we can print the ASCII value of characters in C programming.

```
#include <stdio.h>
void main() { // we are returning void here
// that's why no need of return 0; statement
    char c;
    printf("Enter a character: ");
    scanf("%c", &c);

    // %d displays the integer value of a character
    // %c displays the actual character
    printf("ASCII value of %c = %d", c, c);
}
```

Output:

Enter a character: C  
ASCII value of C = 67

In this program, the user is asked to enter a character. The character is stored in variable c. When %d format string is used, 67 (the ASCII value of C) is displayed. When %c format string is used, 'C' itself is displayed.

C program to find character from ASCII value

```
#include <stdio.h>
void main() { // we are returning void here
// that's why no need of return 0; statement at the end of main function
    int num;
    printf("Enter a Number between 0-255: ");
    scanf("%d", &num);

    // %d displays the integer value of a character
    // %c displays the actual character corresponding to that
    // number
    printf("Character representing %d is %d", num, num);
}
```

Output:

Enter a Number between 0-255: 78  
Character representing 78 is N

## 6. C Program to Compute Quotient and Remainder

In this example, we will learn to find the quotient and remainder when an integer is divided by another integer.

```
#include <stdio.h>
int main() {
    int dividend, divisor, quotient, remainder;
    printf("Enter dividend: ");
    scanf("%d", &dividend);
    printf("Enter divisor: ");
    scanf("%d", &divisor);

    // Computes quotient
    quotient = dividend / divisor;

    // Computes remainder
    remainder = dividend % divisor;

    printf("Quotient = %d\n", quotient);
    printf("Remainder = %d", remainder);
    return 0;
}
```

Output:

Enter dividend: 201

Enter divisor: 10

Quotient = 20

Remainder = 1

In this program, the user is asked to enter two integers (dividend and divisor). They are stored in variables dividend and divisor respectively.

```
printf("Enter dividend: ");
scanf("%d", &dividend);
printf("Enter divisor: ");
scanf("%d", &divisor);
```

Then the quotient is evaluated using / (the division operator), and stored in quotient.

```
quotient = dividend / divisor;
```

Similarly, the remainder is evaluated using % (the modulo operator) and stored in remainder.

```
remainder = dividend % divisor;
```

Finally, the quotient and remainder are displayed using printf().

```
printf("Quotient = %d\n", quotient);
printf("Remainder = %d", remainder);
```

### **Work to do:**

Find out difference between float division and int division in C, use google.

## 7. C Program to Find the Size of int, float, double and char

In this example, you will learn to evaluate the size of each variable using sizeof operator.

```
#include<stdio.h>
int main() {
    int intType;
    float floatType;
    double doubleType;
    char charType;

    // sizeof evaluates the size of a variable
    printf("Size of int: %d bytes\n", sizeof(intType));
    printf("Size of float: %d bytes\n", sizeof(floatType));
    printf("Size of double: %d bytes\n", sizeof(doubleType));
    printf("Size of char: %d byte\n", sizeof(charType));

    return 0;
}
```

Output:

Size of int: 4 bytes

Size of float: 4 bytes

Size of double: 8 bytes

Size of char: 1 byte

In this program, 4 variables intType, floatType, doubleType and charType are declared.

Then, the size of each variable is computed using the sizeof operator.

[Read more about different operators in C](#)

## 8. C Program to Swap Two Numbers

In this example, you will learn to swap two numbers in C programming using two different techniques.

```
#include<stdio.h>
void main() {
    float first, second, temp;
    printf("Enter first number: ");
    scanf("%f", &first);
    printf("Enter second number: ");
    scanf("%f", &second);

    // Value of first is assigned to temp
    temp = first;

    // Value of second is assigned to first
    first = second;

    // Value of temp (initial value of first) is assigned to second
    second = temp;

    printf("\nAfter swapping, firstNumber = %.2f\n", first);
    printf("After swapping, secondNumber = %.2f", second);
}
```

Output:

Enter first number: 14

Enter second number: 34.5

After swapping, firstNumber = 34.50

After swapping, secondNumber = 14.00

In the above program, the temp variable is assigned the value of the first variable. Then, the value of the first variable is assigned to the second variable.

Finally, the temp (which holds the initial value of first) is assigned to second. This completes the swapping process.

### Swap Numbers Without Using Temporary Variables

```
#include <stdio.h>
int main() {
    double a, b;
    printf("Enter a: ");
    scanf("%lf", &a);
    printf("Enter b: ");
    scanf("%lf", &b);

    // Swapping
```

[Author: printf\("Electroverts Lab"\);](#)

```
// a = (initial_a - initial_b)
a = a - b;

// b = (initial_a - initial_b) + initial_b = initial_a
b = a + b;

// a = initial_a - (initial_a - initial_b) = initial_b
a = b - a;

printf("After swapping, a = %.2lf\n", a);
printf("After swapping, b = %.2lf", b);
return 0;
}
```

Output:

Enter a: 14

Enter b: 34.5

After swapping, a = 34.50

After swapping, b = 14.00

## 9. C Program to Check Whether a Number is Even or Odd

In this example, we will learn to check whether a number entered by the user is even or odd.

To understand this example, we should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

An even number is an integer that is exactly divisible by 2. For example: 0, 8, -24

An odd number is an integer that is not exactly divisible by 2. For example: 1, 7, -11, 15

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);

    // True if num is perfectly divisible by 2
    if(num % 2 == 0)
        printf("%d is even.", num);
    else
        printf("%d is odd.", num);

    return 0;
}
```

Output:

Enter an integer: 21

[Author: printf\("Electroverts Lab"\);](#)

21 is odd.

Enter an integer: 20

20 is even.

In the program, the integer entered by the user is stored in the variable num.

Then, whether num is perfectly divisible by 2 or not is checked using the modulus % operator.

If the number is perfectly divisible by 2, test expression `number%2 == 0` evaluates to 1 (true). This means the number is even.

However, if the test expression evaluates to 0 (false), the number is odd.

### Program to Check Odd or Even Using the [Ternary Operator](#)

```
#include <stdio.h>
int main() {
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);

    (num % 2 == 0) ? printf("%d is even.", num) : printf("%d is odd.", num);
    return 0;
}
```

Output:

Enter an integer: 31

31 is odd.

Enter an integer: 32

32 is even.

## [10. C Program to Check Whether a Character is a Vowel or Consonant](#)

In this example, we will learn to check whether an alphabet entered by the user is a vowel or a consonant.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)
- [C while and do...while Loop](#)

The five letters A, E, I, O and U are called vowels. All other alphabets except these 5 vowels are called consonants.



[Author: printf\("Electroverts Lab"\);](#)

This program assumes that the user will always enter an alphabet character.

```
#include <stdio.h>
int main() {
    char c;
    int lowercase_vowel, uppercase_vowel;
    printf("Enter an alphabet: ");
    scanf("%c", &c);

    // evaluates to 1 if variable c is a lowercase vowel
    lowercase_vowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
    // above || means OR, this signifies that if we get any of a, e, i, o, u from
    //scanf as input lowercase_vowel will have "True" stored in it which means 1.

    // evaluates to 1 if variable c is a uppercase vowel
    uppercase_vowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    // evaluates to 1 (true) if c is a vowel
    if (lowercase_vowel || uppercase_vowel) //here if any of the variable is true,
        // that means we have given a true input (Vowel) and we will print the output
        // as vowel
        printf("%c is a vowel.", c);
    else // else we will print the input value is consonant.
        printf("%c is a consonant.", c);
    return 0;
}
```

Output:

Enter an alphabet: A  
A is a vowel.

Enter an alphabet: e  
e is a vowel.

Enter an alphabet: S  
S is a consonant.

Enter an alphabet: h  
h is a consonant.

The character entered by the user is stored in variable c.

The lowercase\_vowel variable evaluates to 1 (true) if c is a lowercase vowel and 0 (false) for any other characters.

Similarly, the uppercase\_vowel variable evaluates to 1 (true) if c is an uppercase vowel and 0 (false) for any other character.

If either lowercase\_vowel or uppercase\_vowel variable is 1 (true), the entered character is a vowel. However, if both lowercase\_vowel and uppercase\_vowel variables are 0, the entered character is a consonant.

**Note:** This program assumes that the user will enter an alphabet. If the user enters a non-alphabetic character, it displays the character is a constant.

[Author: printf\("Electroverts Lab"\);](#)

To fix this, we can use the `isalpha()` function. The **isalpha()** function checks whether a character is an alphabet or not.

```
#include <ctype.h>
#include <stdio.h>

int main() {
    char c;
    int lowercase_vowel, uppercase_vowel;
    printf("Enter an alphabet: ");
    scanf("%c", &c);

    // evaluates to 1 if variable c is a lowercase vowel
    lowercase_vowel = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');

    // evaluates to 1 if variable c is a uppercase vowel
    uppercase_vowel = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    // Show error message if c is not an alphabet
    if (!isalpha(c))
        printf("Error! Non-alphabetic character.");
    else if (lowercase_vowel || uppercase_vowel)
        printf("%c is a vowel.", c);
    else
        printf("%c is a consonant.", c);

    return 0;
}
```

Output:

Enter an alphabet: 11

Error! Non-alphabetic character.

## 11. C Program to Find the Largest Number Among Three Numbers

In this example, we will learn to find the largest number among the three numbers entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

### Example 1: Using if Statement

```
#include <stdio.h>
int main() {
```

[Author: printf\("Electroverts Lab"\);](#)

```
double n1, n2, n3;
printf("Enter three different numbers: ");
scanf("%lf %lf %lf", &n1, &n2, &n3);

// if n1 is greater than both n2 and n3, n1 is the largest
if (n1 >= n2 && n1 >= n3)
    printf("%.2f is the largest number.", n1);

// if n2 is greater than both n1 and n3, n2 is the largest
if (n2 >= n1 && n2 >= n3)
    printf("%.2f is the largest number.", n2);

// if n3 is greater than both n1 and n2, n3 is the largest
if (n3 >= n1 && n3 >= n2)
    printf("%.2f is the largest number.", n3);

return 0;
}
```

Output:

Enter three different numbers: 23

11

26

26.00 is the largest number.

**Quick question...** What will happen if we write %0.0f instead of %.2f in printf statement.

### Example 2: Using if...else Ladder

```
#include <stdio.h>
int main() {
    double n1, n2, n3;
    printf("Enter three numbers: ");
    scanf("%lf %lf %lf", &n1, &n2, &n3);

    // if n1 is greater than both n2 and n3, n1 is the largest
    if (n1 >= n2 && n1 >= n3)
        printf("%.2lf is the largest number.", n1);

    // if n2 is greater than both n1 and n3, n2 is the largest
    else if (n2 >= n1 && n2 >= n3)
        printf("%.2lf is the largest number.", n2);

    // if both above conditions are false, n3 is the largest
    else
        printf("%.2lf is the largest number.", n3);

    return 0;
}
```

Output:

[Author: printf\("Electroverts Lab"\);](#)

Enter three numbers: 23  
27  
31  
31.00 is the largest number.

### Example 3: Using Nested if...else

```
#include <stdio.h>
int main() {
    double n1, n2, n3;
    printf("Enter three numbers: ");
    scanf("%lf %lf %lf", &n1, &n2, &n3);

    if (n1 >= n2) {
        if (n1 >= n3)
            printf("%.2lf is the largest number.", n1);
        else
            printf("%.2lf is the largest number.", n3);
    } else {
        if (n2 >= n3)
            printf("%.2lf is the largest number.", n2);
        else
            printf("%.2lf is the largest number.", n3);
    }

    return 0;
}
```

Output:  
Enter three numbers: 23  
27  
31  
31.00 is the largest number.

So, here we realized that there can be multiple algorithms (methods) to solve a problem.

## 12. C Program to Find the Roots of a Quadratic Equation

In this example, we will learn to find the roots of a quadratic equation in C programming.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

The standard form of a quadratic equation is:

$ax^2 + bx + c = 0$ , where

a, b and c are real numbers and  $a \neq 0$

The term  $b^2 - 4ac$  is known as the discriminant of a quadratic equation. It tells the nature of the roots.

- If the discriminant is greater than 0, the roots are real and different.
- If the discriminant is equal to 0, the roots are real and equal.
- If the discriminant is less than 0, the roots are complex and different.

If the discriminant > 0,	$\text{root1} = \frac{-b + \sqrt{(b - 4ac)}}{2a}$	If the discriminant < 0,	$\text{root1} = \text{root2} = \frac{-b}{2a}$
	$\text{root2} = \frac{-b - \sqrt{(b - 4ac)}}{2a}$		$\text{root1} = \frac{-b}{2a} + i \frac{\sqrt{-(b - 4ac)}}{2a}$ $\text{root2} = \frac{-b}{2a} - i \frac{\sqrt{-(b - 4ac)}}{2a}$

```
#include <math.h>
#include <stdio.h>
int main() {
    double a, b, c, discriminant, root1, root2, realPart, imagPart;
    printf("Enter coefficients a, b and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    discriminant = b * b - 4 * a * c;

    // condition for real and different roots
    if (discriminant > 0) {
        root1 = (-b + sqrt(discriminant)) / (2 * a);
        root2 = (-b - sqrt(discriminant)) / (2 * a);
        printf("root1 = %.2lf and root2 = %.2lf", root1, root2);
    }
```

```
}

// condition for real and equal roots
else if (discriminant == 0) {
    root1 = root2 = -b / (2 * a);
    printf("root1 = root2 = %.2lf;", root1);
}
// if roots are not real
else {
    realPart = -b / (2 * a);
    imagPart = sqrt(-discriminant) / (2 * a);
printf("root1 = %.2lf+%.2lfi and root2 = %.2f-
%.2fi", realPart, imagPart, realPart, imagPart);
}

return 0;
}
```

In this program, the **sqrt()** library function is used to find the square root of a number. The function comes with math.h header file.

Output:

Enter coefficients a, b and c: 4

6

1

root1 = -0.19 and root2 = -1.31

### 13. C Program to Check Leap Year

In this example, we will learn to check whether the year entered by the user is a leap year or not.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400.

For example,

1999 is not a leap year

2000 is a leap year

2004 is a leap year

#### **Program to Check Leap Year**

```
#include <stdio.h>
int main() {
    int year;
```

[Author: printf\("Electroverts Lab"\);](#)

```
printf("Enter a year: ");
scanf("%d", &year);

// leap year if perfectly visible by 400
if (year % 400 == 0) {
    printf("%d is a leap year.", year);
}
// not a leap year if visible by 100
// but not divisible by 400
else if (year % 100 == 0) {
    printf("%d is not a leap year.", year);
}
// leap year if not divisible by 100
// but divisible by 4
else if (year % 4 == 0) {
    printf("%d is a leap year.", year);
}
// all other years are not leap year
else {
    printf("%d is not a leap year.", year);
}
return 0;
}
```

Output:

Enter a year: 2020  
2020 is a leap year.

Enter a year: 2021  
2021 is not a leap year.

## 14. C Program to Check Whether a Number is Positive or Negative

In this example, we will learn to check whether a number (entered by the user) is negative or positive.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

This program takes a number from the user and checks whether that number is either positive or negative or zero.

### Check Positive or Negative Using if...else

```
#include <stdio.h>
int main() {
    double num;
    printf("Enter a number: ");
    scanf("%lf", &num);
    if (num <= 0.0) {
        if (num == 0.0)
            printf("You entered 0.");
        else
            printf("You entered a negative number.");
    } else
        printf("You entered a positive number.");
    return 0;
}
```

Output:

*Enter a number: 99*

You entered a positive number.

*Enter a number: -23*

You entered a negative number.

*Enter a number: 0*

You entered 0.

**We can also solve this problem using nested if else statement.**

### Check Positive or Negative Using Nested if...else

```
#include <stdio.h>
int main() {
    double num;
    printf("Enter a number: ");
    scanf("%lf", &num);

    if (num < 0.0)
        printf("You entered a negative number.");
    else if (num > 0.0)
        printf("You entered a positive number.");
    else
        printf("You entered 0.");

    return 0;
}
```

Output:

*Enter a number: 12*

You entered a positive number.

*Enter a number: -12*



[Author: printf\("Electroverts Lab"\);](#)

You entered a negative number.

Enter a number: 0

You entered 0.

## 15. C Program to Check Whether a Character is an Alphabet or not

In this example, we will learn to check whether a character entered by the user is an alphabet or not.

To understand this example, you should have the knowledge of the following C programming topics:

- [C Programming Operators](#)
- [C if...else Statement](#)

In C programming, a character variable holds an ASCII value (an integer number between 0 and 127) rather than that character itself.

The ASCII value of the lowercase alphabet is from 97 to 122. And, the ASCII value of the uppercase alphabet is from 65 to 90.

If the ASCII value of the character entered by the user lies in the range of 97 to 122 or from 65 to 90, that number is an alphabet.

```
#include <stdio.h>
int main() {
    char c;
    printf("Enter a character: ");
    scanf("%c", &c);

    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))
        printf("%c is an alphabet.", c);
    else
        printf("%c is not an alphabet.", c);

    return 0;
}
```

Output:

Enter a character: E

E is an alphabet.

Enter a character: s

s is an alphabet.

[Author: printf\("Electroverts Lab"\);](#)

**Note:** It is recommended to use the **isalpha()** function to check whether a character is an alphabet or not. Isalpha() is a predefined function.

## 16. C Program to Calculate the Sum of Natural Numbers

In this example, we will learn to calculate the sum of natural numbers entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C for Loop](#)

[C while Loop](#)

The positive numbers 1, 2, 3... are known as natural numbers. The sum of natural numbers up to 10 is:

sum = 1 + 2 + 3 + ... + 10

### Sum of first n Natural Numbers Using for Loop

```
#include <stdio.h>
int main() {
    int n, i, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &n);

    // we are taking input as first n natural numbers we are going to add
    for (i = 1; i <= n; ++i) { // starting from i = 1, we will go on till i <= n and
// we will keep increasing i by 1 after each iteration. In 1st iteration i will be 1, in
2nd i will be 2, in 3rd i will be 3 as so on.
        sum += i; // here we are adding each and every input in variable named sum
    }

    printf("Sum = %d", sum); // finally we are printing sum
    return 0;
}
```

Output:

Enter a positive integer: 10

Sum = 55

Enter a positive integer: 20

Sum = 210

### Sum of first n Natural Numbers Using while Loop

```
#include <stdio.h>
int main() {
```

[Author: printf\("Electroverts Lab"\);](#)

```
int n, i, sum = 0;
printf("Enter a positive integer: ");
scanf("%d", &n); // we are taking input as first n natural numbers we are going to add
i = 1; // we are assigning 1 to variable named i

while (i <= n) { // while i is smaller or equal to n, i <=n will give True
    // if true then the program will enter inside the body of while loop
    sum += i; // we will add i to variable named sum in each iteration
    ++i; // we are increasing i by 1
}

printf("Sum = %d", sum); // finally we are printing total sum
return 0;
}
```

Output:

Enter a positive integer: 10  
Sum = 55

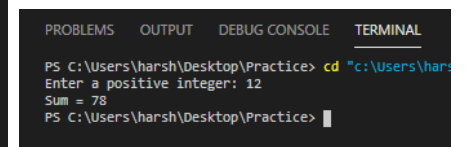
Enter a positive integer: 20  
Sum = 210

Okay, we can calculate above sum of n natural numbers by formula too.

Sum of n natural numbers =  $n(n+1)/2$ .

Let's code:

```
1 // Program to calculate the sum of first n natural numbers
2 // Positive integers 1,2,3...n are known as natural numbers
3 #include <stdio.h>
4 int main()
5 {
6     int num, sum;
7     printf("Enter a positive integer: ");
8     scanf("%d", &num); // taking the integer type input
9     // and storing it in num variable that we created above
10
11     // now applying formula of n(n+1)/2
12     sum = num * (num + 1) / 2;
13
14     printf("Sum = %d", sum);
15
16     return 0;
17 }
```



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\Users\harsh\Desktop\Practice> cd "c:\Users\harsh\Desktop\Practice"
Enter a positive integer: 12
Sum = 78
PS C:\Users\harsh\Desktop\Practice>
```

## 17. C Program to Find Factorial of a Number

In this example, we will learn to calculate the factorial of a number entered by the user.

To understand this example, we should have the knowledge of the following C programming topics:

[C Data Types](#)

[C Programming Operators](#)

[C if...else Statement](#)

[C for Loop](#)

The factorial of a positive number  $n$  is given by:

Factorial of  $n$  ( $n!$ ) =  $1 * 2 * 3 * 4 * \dots * n$

The factorial of a negative number doesn't exist. And, the factorial of 0 is 1.

### Factorial of a Number using C programming

```
#include <stdio.h>
int main() { // return type is int so we will be expecting main function to return an
// integer at the end
    int n, i;
    unsigned long fact = 1; // we can take unsigned datatype here because we are
// sure that factorial of a number can't be negative
    printf("Enter an integer: ");
    scanf("%d", &n); // enter the number whose factorial we want to get

    // shows error if the user enters a negative integer
    if (n < 0)
        printf("Error! Factorial of a negative number doesn't exist.");
    else { // now we will run a loop from i to n where we will keep multiplying
// i by i+1 in each iteration till the value of i reaches n
        for (i = 1; i <= n; ++i) {
            fact *= i; // fact = fact * i;
        }
        printf("Factorial of %d = %lu", n, fact); // factorial of the number is stored in
// variable named fact, we are printing that finally here
    }

    return 0; // returning integer at end as we wrote main function will return integer value
}
```

[Author: printf\("Electroverts Lab"\);](#)

Output:

Enter an integer: 5

Factorial of 5 = 120

Enter an integer: 10

Factorial of 10 = 3628800

## 18. C Program to Generate Multiplication Table

In this example, we will learn to generate the multiplication table of a number entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C for Loop](#)

The program below takes an integer input from the user and generates the multiplication tables up to 10.

### **Multiplication Table Up to 10**

```
#include <stdio.h>
int main() {
    int n, i;
    printf("Enter an integer: ");
    scanf("%d", &n); // take a number as input from user
    for (i = 1; i <= 10; ++i) { // run a loop from i = 1 to i = 10, loop will iterate
        10 times
        printf("%d * %d = %d \n", n, i, n * i); // print n * i = n*i where I will keep
        increasing
        // from 1 to 10, also give newline after each multiplication
    }
    return 0;
}
```

Output:

Enter an integer: 19

19 \* 1 = 19

19 \* 2 = 38

19 \* 3 = 57

19 \* 4 = 76

19 \* 5 = 95

19 \* 6 = 114

19 \* 7 = 133

19 \* 8 = 152

[Author: printf\("Electroverts Lab"\);](#)

19 \* 9 = 171  
19 \* 10 = 190

Here's a little modification of the above program to generate the multiplication table up to a range (where range is also a positive integer entered by the user). Previously we were doing it for 10 times only.

### Multiplication Table Up to a range

```
#include <stdio.h>
int main() {
    int n, i, range;
    printf("Enter an integer: ");
    scanf("%d", &n);
    printf("Enter the range: ");
    scanf("%d", &range);
    for (i = 1; i <= range; ++i) { // here everything is same except that
        // we are not going from i = 1 to i = 10, instead we are going from i = 1 to i =
range
        // entered by the user
        printf("%d * %d = %d \n", n, i, n * i);
    }
    return 0;
}
```

Output:

Enter an integer: 19

Enter the range: 15

19 \* 1 = 19  
19 \* 2 = 38  
19 \* 3 = 57  
19 \* 4 = 76  
19 \* 5 = 95  
19 \* 6 = 114  
19 \* 7 = 133  
19 \* 8 = 152  
19 \* 9 = 171  
19 \* 10 = 190  
19 \* 11 = 209  
19 \* 12 = 228  
19 \* 13 = 247  
19 \* 14 = 266  
19 \* 15 = 285

## 19. C Program to Display Fibonacci sequence

In this example, we will learn to display the Fibonacci sequence of first n numbers (entered by the user).

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C while and do...while Loop](#)

[C for Loop](#)

[C break and continue](#)

The Fibonacci sequence is a sequence where the next term is the sum of the previous two terms. The first two terms of the Fibonacci sequence are 0 followed by 1.

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21

### **Fibonacci Series up to n terms using for Loop**

```
/* The Fibonacci sequence is a sequence where the next term is the sum of the previous two terms.
The first two terms of the Fibonacci sequence are 0 followed by 1.
The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21
*/

#include <stdio.h>
int main() {
    int i, n, t1 = 0, t2 = 1, nextTerm;
    printf("Enter the number of terms: ");
    scanf("%d", &n); // taking input for number of terms we want in the series
    printf("Fibonacci Series: ");

    for (i = 1; i <= n; ++i) { // starting the loop from i = 1 to i<=n i.e. loop will run n number of times
        printf("%d, ", t1); // first we will print value of t1, i.e. 0
        nextTerm = t1 + t2; // then we will add t1 & t2 and store that in a variable called nextTerm
        t1 = t2; // we will update t1 with t2
        t2 = nextTerm; // we will update t2 with sum of t1 & t2 i.e. nextTerm
    }
    printf("\b\b ");
    // then the loop will run again. Taking example, in 1st loop t1 = 0 and t2 = 1,
    // nextTerm = 1+0 = 1, then we will save t1=t2, t1 will become 1 and t2 will become 1
}
```

[Author: printf\("Electroverts Lab"\);](#)

```
// then next loop will start, we will print t1 i.e. 1, after that we will again add t1 and t2
// that will give us 1 + 1 = 2, so 2 will be saved in nextTerm and 1 will again be saved in t1
// after that we are assigning t1 = t2, so t1 will become 1 also, t2 = nextTerm, t2 will become 2.

return 0;
}
```

Output:

*Enter the number of terms: 10*

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

We have “ , ” coming at the end of 34 because when we are in last loop then comma along with last digit is printed.

Now how can I remove that last comma?

We will use “\b” [escape sequence](#) here.

Just add

```
printf("\b\b ");
```

Below the loop and it will work as backspace where 1<sup>st</sup> \b will be used to remove extra space and 2<sup>nd</sup> \b will be used to remove comma after 34.

Output:

*Enter the number of terms: 10*

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

## [20. C Program to Display Characters from A to Z Using Loop](#)

In this example, we will learn to print all the letters of the English alphabet.

To understand this example, you should have the knowledge of the following C programming topics:

[C if...else Statement](#)

[C while and do...while Loop](#)

### **Program to Print English Alphabets**

```
#include <stdio.h>
int main() {
    char c; // we are first making one variable of character type named char
```



[Author: printf\("Electroverts Lab"\);](#)

```
    for (c = 'A'; c <= 'Z'; ++c) // here we are starting from A and going till Z , al
so inreasing
    // the value of c by 1 after each iteration. A = 65 in ASCII and Z = 90
    // similarly a = 97 & z = 122
    printf("%c ", c);
    return 0;
}
```

Output:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Now we can use ASCII code too for printing alphabets as we already know that A= 65 & Z = 90

```
#include <stdio.h>
int main() {
    char c; // we are first making one variable of character type nemed char

    for (c = 65; c <= 90; ++c) // here we are starting from 65 and going till 90, als
o inreasing
    // the value of c by 1 after each iteration.

    printf("%c ", c); // we are printing cahacter of 65 in 1st iteration i.e. A
    // in 2nd interartion c will become 66 and that will resemble B in ASCII
    return 0;
}
```

Output:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Similarly we can write

for (c = 97; c<=122; c++) to print small alphabets

```
#include <stdio.h>
int main() {
    char c; // we are first making one variable of character type nemed char

    for (c = 97; c<=122; c++) // here we are starting from 97 and going till 122,
also inreasingthe value of c by 1 after each iteration.

    printf("%c ", c); // we are printing cahacter of 65 in 1st iteration i.e. A
    // in 2nd interartion c will become 66 and that will resemble B in ASCII
    return 0;
}
```

Output:

a b c d e f g h i j k l m n o p q r s t u v w x y z

## 21. C Program to Count Number of Digits in an Integer

In this example, we will learn to count the number of digits in an integer entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C while Loop](#)

This program takes an integer from the user and calculates the number of digits. For example: If the user enters 2319, the output of the program will be 4.

### **Program to Count the Number of Digits**

```
#include <stdio.h>
int main() {
    long long n;
    int count = 0;
    printf("Enter an integer: ");
    scanf("%lld", &n);

    // remove last digit from n in each iteration
    // increase count variable i.e. 0 by 1 in each iteration
    while (n != 0) {
        n /= 10;    // n = n/10, let's suppose that n = 123
        // n = n/10 will give 12. Again, 12/10 will give 1 that is quotient
        ++count;
        // iterate until n becomes 0
    }

    printf("Number of digits: %d", count);
}
```

Output:

Enter an integer: 123456

Number of digits: 6

The integer entered by the user is stored in variable n. Then the while loop is iterated until the test expression  $n \neq 0$  is evaluated to 0 (false).

After the first iteration, the value of n will be 12345 and the count is incremented to 1.  
After the second iteration, the value of n will be 1234 and the count is incremented to 2.  
After the third iteration, the value of n will be 123 and the count is incremented to 3.  
After the fourth iteration, the value of n will be 12 and the count is incremented to 4.  
After the fifth iteration, the value of n will be 1 and the count is incremented to 5.  
After the sixth iteration, the value of n will be 0 and the count is incremented to 6.

Then the test expression of the loop is evaluated to false and the loop terminates.

## 22. C Program to Reverse a Number

In this example, we will learn to reverse the number entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C while Loop](#)

### **Reverse an Integer**

```
#include <stdio.h>
int main() {
    int n, rev = 0, remainder;
    printf("Enter an integer: ");
    scanf("%d", &n);
    while (n != 0) {
        remainder = n % 10; // % is Modulo operator which will give remainder
        // so here n % 10 will give remainder which will be eventually saved in
        remainder variable
        // n % 10 will basically give last digit of the number n
        rev = rev * 10 + remainder; // so we have last digit of the number n,
        now we have a variable named rev
        // where we will be storing reversed number, in 1st loop rev = 0 so
        rev = 0 * 10 + remainder
        // In 2nd loop rev = 1st-rev * 10 + 2nd-remainder
        // in 3rd loop rev = 2nd-
        rev * 10 + 3rd remainder and the loop will go on till n becomes 0

        n /= 10; // this says that n = n/10, in C programming n/10 will mean that
        last term of n will be gone because
        // we are doing a integer division and here 55/10 will be 5 not 5.5, even
        59/10 will give 5 only, because it is taking floor number by default
    }
    printf("Reversed number = %d", rev);
    return 0;
}
```

Output:

Enter an integer: 12345

Reversed number = 54321

[Concept of floor and ceil in C programming](#)

## 23. C Program to Calculate the Power of a Number

In this example, we will learn to calculate the power of a number.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C while Loop](#)

The program below takes two integers from the user (a base number and an exponent) and calculates the power.

For example: In the case of 2 as base & 3 as power

2 is the base number

3 is the exponent

And, the power is equal to  $2 \times 2 \times 2$

### **Power of a Number Using the while Loop**

```
#include <stdio.h>
int main() {
    int base, exp; // variable for taking input for base and power
    long long result = 1; // variable for storing results
    printf("Enter a base number: ");
    scanf("%d", &base); // taking input for base
    printf("Enter an exponent: ");
    scanf("%d", &exp); // taking input for power/ exponent

    while (exp != 0) { // we basically will multiply base power number of times
        result *= base; // if base is 2 and power is 4, the loop will run 4 times multiplying
        // 2 in the result variable in every loop in that way after 4 loops, we will get 2**4
        --exp;
    }
    printf("Answer = %lld", result);
    return 0;
}
```

Output:

Enter a base number: 3

Enter an exponent: 4

Answer = 81

The above technique works only if the exponent is a positive integer.

If you need to find the power of a number with any real number as an exponent, you can use the **pow()** function.

### Power Using pow() Function

```
#include <math.h>
#include <stdio.h>

int main() {
    double base, exp, result;
    printf("Enter a base number: ");
    scanf("%lf", &base);
    printf("Enter an exponent: ");
    scanf("%lf", &exp);

    // calculates the power
    result = pow(base, exp);

    printf("%.11f^%.11f = %.21f", base, exp, result);
    return 0;
}
```

Output:

Enter a base number: 3

Enter an exponent: 4

Answer = 81

## 24. C Program to Check Whether a Number is Palindrome or Not

In this example, we will learn to check whether the number entered by the user is a palindrome or not.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C if...else Statement](#)

[C while Loop](#)

An integer is a palindrome if the reverse of that number is equal to the original number.

MOM is a palindrome word.

1221 is a palindrome integer.

### Program to Check Palindrome of integer

```
#include <stdio.h>
void main() {
    int n, reversedN = 0, remainder, originalN;
    printf("Enter an integer: ");
    scanf("%d", &n);
    originalN = n; // taking input and saving that in originalN variable

    // reversed integer is stored in reversedN
    while (n != 0) {
        remainder = n % 10; // this will give last digit if the number
        reversedN = reversedN * 10 + remainder;
        n /= 10;
    } // basically from the above loop we are reversing the original number and
        saving that in reversedN variable

    // palindrome if originalN and reversedN are equal
    if (originalN == reversedN) // checking if originalN number and reversedN
        //number are same
        printf("%d is a palindrome.", originalN);
    else
        printf("%d is not a palindrome.", originalN);
}
```

Output:

Enter an integer: 1221  
1221 is a palindrome.

Enter an integer: 1231  
1231 is not a palindrome.

## [25. C Program to Check Whether a Number is Prime or Not](#)

In this example, we will learn to check whether an integer entered by the user is a prime number or not.

To understand this example, we should have the knowledge of the following C programming topics:

[C if...else Statement](#)

[C for Loop](#)

[C break and continue](#)

A prime number is a positive integer that is divisible only by 1 and itself. For example: 2, 3, 5, 7, 11, 13, 17...

```
#include <stdio.h>
int main() {
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);

    for (i = 2; i <= n / 2; ++i) { // we will check for a number is prime or not by
        dividing if from 2
        // to number/2, if the given number is divisible once, the number won't be prime.
        // condition for non-prime
        if (n % i == 0) {
            flag = 1; // if statement if (n%i == 0) is true, that means n is
            totally divided by i which goes from i = 2 to i = n/2
            break; // after that being true, program comes in if statement and makes
            flag = 1 which was 0 initially.
            // Also break statement is executed so that the control comes out of the
            loop as the number is already non-prime
        }
    }

    if (n == 1) {
        printf("1 is neither prime nor composite.");
    }
    else {
        if (flag == 0) // flag is 0 initially, it will only become 1 if the number is
        non-prime. Else, the number is prime only.
            printf("%d is a prime number.", n);
        else
            printf("%d is not a prime number.", n);
    }

    return 0;
}
```

Output:

Enter a positive integer: 31

31 is a prime number.

Enter a positive integer: 20

20 is not a prime number.

Enter a positive integer: 15

15 is not a prime number.

In the program, a for loop is iterated from  $i = 2$  to  $i < n/2$ .

In each iteration, whether  $n$  is perfectly divisible by  $i$  is checked using:

```
if (n % i == 0) {
```

```
}
```

[Author: printf\("Electroverts Lab"\);](#)

If n is perfectly divisible by i, n is not a prime number. In this case, flag is set to 1, and the loop is terminated using the break statement.

After the loop, if n is a prime number, flag will still be 0. However, if n is a non-prime number, flag will be set to 1.

## 26. C Program to Display Prime Numbers between Two Intervals

In this example, we will learn to print all prime numbers between two numbers entered by the user.

To understand this example, we should have the knowledge of the following C programming topics:

[C if...else Statement](#)

[C for Loop](#)

[C break and continue](#)

### Display Prime Numbers between Two Intervals

```
#include <stdio.h>

int main() {
    int low, high, i, flag;
    printf("Enter two numbers(intervals): ");
    scanf("%d %d", &low, &high);
    printf("Prime numbers between %d and %d are: ", low, high);

    // iteration until low is not equal to high
    while (low < high) {
        flag = 0; // flag is put as 0 here
        // let's understand how we are going to solve this problem.
        // Step1: We will run a loop from low to high value entered by the user
        // Step2: If low is less than 1, we will increase the value of low by 1 and run
        //         continue statement
        // Step3: Now we will check if low is prime or not, if prime, we will print that
        //         particular low value and finally increase low by 1
        // step4: Again in new loop value of flag is set to 0, we will check if our low+1
        //         is prime or not, if not-prime flag will become 1
        // and loop will continue without printing anything
        // ignore numbers less than 2
        if (low <= 1) {
            ++low;
            continue;
        }

        // if low is a non-prime number, flag will be 1
```



```
for (i = 2; i <= low / 2; ++i) {  
  
    if (low % i == 0) {  
        flag = 1;  
        break;  
    }  
}  
  
if (flag == 0)  
    printf("%d ", low);  
  
// to check prime for the next number  
// increase low by 1  
++low;  
}  
  
return 0;  
}
```

Output:

Enter two numbers (intervals): 0

20

Prime numbers between 0 and 20 are: 2 3 5 7 11 13 17 19

Enter two numbers (intervals): 100

200

Prime numbers between 100 and 200 are: 101 103 107 109 113 127 131 137 139 149 151 157 163  
167 173 179 181 191 193 197 199

In this program, the while loop is iterated (high-low-1) times.

In each iteration, whether low is a prime number or not is checked, and the value of low is incremented by 1 until low is equal to high.

### Display Prime Numbers when Larger Number is entered first

```
#include <stdio.h>  
  
int main() {  
    int low, high, i, flag, temp;  
    printf("Enter two numbers(intervals): ");  
    scanf("%d %d", &low, &high);  
  
    // swap numbers if low is greather than high  
    if (low > high) { // if 1st number entered is greater than 2nd by mistake we can j  
ust swiipe them  
        temp = low;  
        low = high;  
        high = temp;  
    }
```

[Author: printf\("Electroverts Lab"\);](#)

```
}  
// rest are all same as above  
printf("Prime numbers between %d and %d are: ", low, high);  
while (low < high) {  
    flag = 0;  
  
    // ignore numbers less than 2  
    if (low <= 1) {  
        ++low;  
        continue;  
    }  
  
    for (i = 2; i <= low / 2; ++i) {  
        if (low % i == 0) {  
            flag = 1;  
            break;  
        }  
    }  
    if (flag == 0)  
        printf("%d ", low);  
    ++low;  
}  
  
return 0;  
}
```

Output:

Enter two numbers(intervals): 100

0

Prime numbers between 0 and 100 are: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Enter two numbers(intervals): 0

100

Prime numbers between 0 and 100 are: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

## 27. C Program to Check Armstrong Number

In this example, we will learn to check whether an integer entered by the user is an Armstrong number or not.

To understand this example, you should have the knowledge of the following C programming topics:

[C if...else Statement](#)

[Author: printf\("Electroverts Lab"\);](#)

## [C while Loop](#)

A positive integer is called an Armstrong number (of order n) if:

$$abcd... = a^n + b^n + c^n + d^n +$$

In the case of an Armstrong number of 3 digits, the sum of cubes of each digit is equal to the number itself. For example, 153 is an Armstrong number because

$$153 = 1*1*1 + 5*5*5 + 3*3*3$$

### Check Armstrong Number of three digits

```
#include <stdio.h>
int main() {
    int num, originalNum, remainder, result = 0;
    printf("Enter a three-digit integer: ");
    scanf("%d", &num);
    originalNum = num;
    // let's suppose that our entered number is abc. So, here we will check if
    // a3 + b3 + c3 = abc (entered number)
    while (originalNum != 0) {
        // remainder contains the last digit
        remainder = originalNum % 10;

        result += remainder * remainder * remainder;

        // removing last digit from the original number
        originalNum /= 10; // it will basically take away last digit from number i.e. 1
        // 23/10 = 12
    } // here we are running a loop 3 times, in 1st loop c will be saved in remainder
    // and after cubing it,
    // c3 will be added to results variable, in 2nd loop b will be saved in remainder
    // and we will cube it and add
    // that to results variable where c3 is already existing. Again in last loop, a3
    // will be added to results variable
    // in this way cube of each of the numbers are added in results and if results =
    // num entered by the user
    // the number is Armstrong number

    if (result == num)
        printf("%d is an Armstrong number.", num);
    else
        printf("%d is not an Armstrong number.", num);

    return 0;
}
```

Output:

[Author: printf\("Electroverts Lab"\);](#)

Enter a three-digit integer: 371

371 is an Armstrong number.

### Check Armstrong Number of n digits

```
#include <math.h>
#include <stdio.h>

int main() {
    int num, originalNum, remainder, n = 0;
    float result = 0.0;

    printf("Enter an integer: ");
    scanf("%d", &num);

    originalNum = num;

    // in this problem we will first try to know number of digits in the number
    // provided by the user
    // we will run a loop, either for or while to check number of digits in the
    // provided number

    // store the number of digits of num in n which is initially 0
    while (originalNum != 0) {
        originalNum /= 10;
        ++n;
    }

    // we can also do above activity using for loop as
    /* for (originalNum = num; originalNum != 0; ++n) {
        originalNum /= 10;
    } */

    for (originalNum = num; originalNum != 0; originalNum /= 10) {
        remainder = originalNum % 10;

        // store the sum of the power of individual digits in result
        result += pow(remainder, n);
    }

    // originalNum = num;
    // while (originalNum != 0){
    //     remainder = originalNum % 10; // this will give last digit of originalNum

    //     result = result + pow(remainder,n); // we are using pow function here
    //     to calculate power of a number
    //     // 1st argument takes the base number and 2nd argument takes power }

    // if num is equal to result, the number is an Armstrong number
    if ((int)result == num)
```

[Author: printf\("Electroverts Lab"\);](#)

```
printf("%d is an Armstrong number.", num);  
else  
    printf("%d is not an Armstrong number.", num);  
return 0;  
}
```

Output:

Enter an integer: 407

407 is an Armstrong number.

Enter an integer: 1634

1634 is an Armstrong number.

In this program, the number of digits of an integer is calculated first and stored in n. And, the **pow()** function is used to compute the power of individual digits in each iteration of the second for loop.

## [28. C Program to Display Factors of a Number](#)

In this example, we will learn to find all the factors of an integer entered by the user.

To understand this example, we should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C if...else Statement](#)

[C for Loop](#)

This program takes a positive integer from the user and displays all the positive factors of that number.

### Factors of a Positive Integer

```
#include <stdio.h>  
void main() {  
    int num, i;  
    printf("Enter a positive integer: ");  
    scanf("%d", &num);  
    printf("Factors of %d are: ", num);  
    for (i = 1; i <= num; ++i) { // we will run a loop from i = 1 to the number  
        provided  
  
        if (num % i == 0) { // for each value of i, we will check if that divided  
            provided number completely or not
```

[Author: printf\("Electroverts Lab"\);](#)

```
        printf("%d ", i); // if that particular i divides provided number, i is
        factor of input number and printed so
    }
}}
```

Output:

*Enter a positive integer: 20*

Factors of 20 are: 1 2 4 5 10 20

*Enter a positive integer: 100*

Factors of 100 are: 1 2 4 5 10 20 25 50 100

## 29. C Program to Find GCD of two Numbers

Examples on different ways to calculate GCD of two integers (for both positive and negative integers) using loops and decision making statements.

To understand this example, we should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C for Loop](#)

[C if...else Statement](#)

[C while Loop](#)

The HCF or GCD of two integers is the largest integer that can exactly divide both numbers (without a remainder).

There are many ways to find the greatest common divisor in C programming.

### **Example #1: GCD Using for loop and if Statement**

```
#include <stdio.h>
void main()
{
    int n1, n2, i, gcd;

    printf("Enter two integers: ");
    scanf("%d %d", &n1, &n2);
    // GCD is greatest common divisor, so we have to find greatest number which will
    //divide both input numbers perfectly
```

```
for(i=1; i <= n1 && i <= n2; ++i) // we are running a loop from 1 to
    //i<=n1 && i<=n2 because if i >n1 or n2,
    // it won't be able to divide that number perfectly.
{
    // Checks if i is factor of both integers
    if(n1%i==0 && n2%i==0) // if i will divide both n1 & n2, execution will go
    //inside if statement and
        gcd = i; // i will be saved in gcd variable, loop will run again and if
    // different value of i i.e. i + n will divide
    // both n1 & n2, pervious value of gcd will be replaced by new value of
    // i + n. Loop will run till last and greatest value of i
    // that will divide both n1 && n2 will be saved in gcd. That number will be
    // greatest common divisor of n1 & n2
}

printf("G.C.D of %d and %d is %d", n1, n2, gcd);
}
```

Output:

Enter two integers: 20

50

G.C.D of 20 and 50 is 10

Enter two integers: 60

100

G.C.D of 60 and 100 is 20

In this program, two integers entered by the user are stored in variable n1 and n2. Then, for loop is iterated until i is less than n1 and n2.

In each iteration, if both n1 and n2 are exactly divisible by i, the value of i is assigned to gcd.

When the for loop is completed, the greatest common divisor of two numbers is stored in variable gcd.

### Example #2: GCD Using while loop and if...else Statement

```
#include <stdio.h>
void main()
{
    int n1, n2;

    printf("Enter two positive integers: ");
    scanf("%d %d",&n1,&n2);
    /* This is a better way to find the GCD. In this method, smaller integer is subtracted
    from the larger integer,
    and the result is assigned to the variable holding larger integer. This process is continued
    until n1 and n2 are equal. */
    while(n1!=n2)
    {
        if(n1 > n2)
            n1 -= n2;
```

[Author: printf\("Electroverts Lab"\);](#)

```
        else
            n2 -= n1;
    }
    printf("GCD = %d",n1);
}
```

Output:

Enter two positive integers: 100  
12  
GCD = 4

Enter two positive integers: 50  
80  
GCD = 10

The above two programs works as intended only if the user enters positive integers. Here's a little modification of the second example to find the GCD for both positive and negative integers.

### Example #3: GCD for both positive and negative numbers

```
#include <stdio.h>
int main()
{
    int n1, n2;

    printf("Enter two integers: ");
    scanf("%d %d",&n1,&n2);

    // if user enters negative number, sign of the number is changed to positive
    n1 = ( n1 > 0 ) ? n1 : -
n1; // if the number is negative, number is changed to positive
    n2 = ( n2 > 0 ) ? n2 : -n2;

    // then same process of finding gcd is repeated again
    while(n1!=n2)
    {
        if(n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    }
    printf("GCD = %d",n1);

    return 0;
}
```

Output:

Enter two integers: -10  
-25  
GCD = 5



[Author: printf\("Electroverts Lab"\);](#)

Enter two integers: -60

72

GCD = 12

### [30. C Program to Find LCM of two Numbers](#)

In this example, we will learn to calculate the LCM (Lowest common multiple) of two numbers entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C if...else Statement](#)

[C while Loop](#)

The LCM of two integers n1 and n2 is the smallest positive integer that is perfectly divisible by both n1 and n2 (without a remainder). For example, the LCM of 72 and 120 is 360.

#### LCM using while and if

```
#include <stdio.h>
void main() {
    int n1, n2, max;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);

    // Least Common Multiple, here we will find a number say max which will be perfectly divided by n1 & n2 both
    // and condition is that, max should be least common multiple number. i.e Common multiple of 12 & 18 can be
    // both 36 & 72, but LCM will be 36 only because it is least common multiple

    // maximum number between n1 and n2 is stored in max
    max = (n1 > n2) ? n1 : n2;

    while (1) {
        if (max % n1 == 0 && max % n2 == 0) {
            printf("The LCM of %d and %d is %d.", n1, n2, max);
            break;
        }
        ++max;
    }
}
```

Output:

[Author: printf\("Electroverts Lab"\);](#)

Enter two positive integers: 12

18

The LCM of 12 and 18 is 36.

In this program, the integers entered by the user are stored in variable n1 and n2 respectively. The largest number among n1 and n2 is stored in min. The LCM of two numbers cannot be less than min.

The test expression of while loop is always true.

In each iteration, whether min is perfectly divisible by n1 and n2 is checked.

```
if (min % n1 == 0 && min % n2 == 0) { ... }
```

If this test condition is not true, min is incremented by 1 and the iteration continues until the test expression of the if statement is true.

The LCM of two numbers can also be found using the formula:

**LCM = (num1\*num2)/GCD**

### LCM Calculation Using GCD

```
#include <stdio.h>
int main() {
    int n1, n2, i, gcd, lcm;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);

    for (i = 1; i <= n1 && i <= n2; ++i) {

        // check if i is a factor of both integers
        if (n1 % i == 0 && n2 % i == 0)
            gcd = i;
    }
    // here we are calculating GCD first and then by given formula, we are finally calculating LCM
    lcm = (n1 * n2) / gcd;

    printf("The LCM of two numbers %d and %d is %d.", n1, n2, lcm);
    return 0;
}
```

Output:

Enter two positive integers: 12

18

The LCM of two numbers 12 and 18 is 36.

[Author: printf\("Electroverts Lab"\);](#)

Enter two positive integers: 36

100

The LCM of two numbers 36 and 100 is 900.

## 31. C Program to Print Pyramids and Patterns

In this example, we will learn to print half pyramids, inverted pyramids, full pyramids, inverted full pyramids, Pascal's triangle, and Floyd's triangle in C Programming.

To understand this example, we should have the knowledge of the following C programming topics:

[C if...else Statement](#)

[C for Loop](#)

[C while Loop](#)

[C break and continue](#)

### Example 1: Half Pyramid of \*

```
*
* *
* * *
* * * *
* * * * *
```

### C Program

```
#include <stdio.h>
int main() {
    int i, j, rows;
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    // Half pyramid will look as right angled triangle where top most row have
    // 1 * and that will keep increasing
    // while we climb down.
    for (i = 1; i <= rows; ++i) { // 1st loop will be for number of rows of asterisk
        // and 2nd loop will be for number of columns
        for (j = 1; j <= i; ++j) { // for each value of i, number of asterisk will keep
            on increasing at uniform rate, 1 in 1st row, 2 in 2nd, 3 in 3rd and so on.
            printf("* ");
        }
        printf("\n"); // after completion of each outer loop, cursor is shifted to newl
        ine so that new row can be written
    }
}
```

[Author: printf\("Electroverts Lab"\);](#)

```
}  
    return 0;  
}
```

Output:

*Enter the number of rows: 5*

```
*  
**  
***  
****  
*****
```

### Example 2: Half Pyramid of Numbers

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

### C Program

```
#include <stdio.h>  
int main() {  
    int i, j, rows;  
    printf("Enter the number of rows: ");  
    scanf("%d", &rows);  
  
    // it will be same code as half pyramid of asterik, just in places where we were p  
    utting asterik, we will put value of j  
    for (i = 1; i <= rows; ++i) { // 1st loop will be for number of rows and 2nd loop  
    will be for number of columns  
        for (j = 1; j <= i; ++j) { // for each value of i, number of digits will keep on  
        increasing at uniform rate, 1 in 1st row, 2 in 2nd, 3 in 3rd and so on.  
            printf("%d ", j);  
        }  
        printf("\n"); // after completion of each outer loop, cursor is shifted to newl  
        ine so that new row can be written  
    }  
    return 0;  
}
```

Output:

*Enter the number of rows: 5*

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

### Example 3: Half Pyramid of Alphabets

```
A
B B
C C C
D D D D
E E E E E
```

#### C Program

```
#include <stdio.h>
int main() {
    int i, j;
    char input, alphabet = 'A';
    // it is similar to half pyramid of asterik, just in places where we were printing
    asterik, we will printing alphabet
    printf("Enter an uppercase character you want to print in the last row: ");
    scanf("%c", &input);
    //suppose input is E
    // here i will run for 69-65+1=5 because ascii of E is 69 and A is 65.
    // j will run uptill i, so that printing values will be equal to number of rows.
    for (i = 1; i <= (input - 'A' + 1); ++i) {
        for (j = 1; j <= i; ++j) {
            printf("%c ", alphabet); //it will print the alphabet i no. of times
        }
        ++alphabet; //incrementing the value of alphabet when j loop finishes
        printf("\n"); //next alphabet should be printed in next line
    }
    return 0;
}
```

Output:

*Enter an uppercase character you want to print in the last row: H*

```
A
B B
C C C
D D D D
E E E E E
F F F F F
G G G G G G
H H H H H H H H
```

### Example 4: Inverted half pyramid of \*

```
* * * * *
```

[Author: printf\("Electroverts Lab"\);](#)

```
* * * *  
* * *  
* *  
*
```

### C Program

```
#include <stdio.h>  
int main() {  
    int i, j, rows;  
    printf("Enter the number of rows: ");  
    scanf("%d", &rows);  
    // for printing an inverted pyramid we will use same logic as half pyramid but the  
    // difference will be rather than  
    // starting from minimum value we have to start from max value to print an inverted  
    // pyramid.  
    for (i = rows; i >= 1; --  
i) { // we are starting loop from max value of i and reducing that by 1 in each loop  
        for (j = 1; j <= i; ++j) { //j will run i th no. of times  
            printf("* "); //asterisk will be printed i th times, for first time it  
            //will be printed max time  
            //and hence reducing by 1 each time j loop is terminated.  
        }  
        printf("\n"); // this avoids printing the next values in the same line to make  
        // a pattern  
    }  
    return 0;  
}
```

Output:

*Enter the number of rows: 5*

```
* * * * *  
* * * *  
* * *  
* *  
*
```

### Example 5: Inverted half pyramid of numbers

```
1 2 3 4 5  
1 2 3 4  
1 2 3  
1 2  
1
```

### C Program

```
#include <stdio.h>  
int main() {  
    int i, j, rows;  
    printf("Enter the number of rows: ");
```

```
scanf("%d", &rows);
// for printing an inverted pyramid of numbers, we will use same logic as half pyramid,
// printing from 1 to 5 then 1 to 4 in next line and so on.
for (i = rows; i >= 1; --i) { // we are starting loop from max value of i and reducing that by 1 in each loop
    for (j = 1; j <= i; ++j) {
        printf("%d ", j); // values of j will be printed starting from 1 to the value of i
    }
    printf("\n"); // brings the cursor to the next line so that next set of values of j should be printed in the next line.
}
return 0;
}
```

Output:

Enter the number of rows: 5

1 2 3 4 5

1 2 3 4

1 2 3

1 2

1

#### Example 6: Full Pyramid of \*

```
      *
    * * *
  * * * * *
*****
*****
*****
```

#### C Program

```
#include <stdio.h>
int main() {
    int i, space, rows, k = 0;
    printf("Enter the number of rows: ");
    // here we have to analyze the question, we have 1 asterik in 1st row, 3 in 2nd 5 in 3rd..so it follows a sequence of 2n-1
    // We have 4 blank spaces in 1st row, 3 in 2nd, 2 in 3rd, so it follows pattern of 5-n where 5 is total number of rows in n is nth row

    scanf("%d", &rows);
    for (i = 1; i <= rows; ++i, k = 0) { // in outer for loop we are initializing i=1 and keeping i<=rows and increasing i by 1 in each loop..also,
        // we are initializing another variable k=0 which we are going to use later in the loop
    }
```

[Author: printf\("Electroverts Lab"\);](#)

```
    for (space = 1; space <= rows -
i; ++space) { // now 1st we are printing space following same pattern of total_rows -
i th row.
        printf(" "); // this loop will print 4 spaces in 1st row, 3 spaces in 2nd a
nd so on
    }
    while (k != 2 * i -
1) { // after printing spaces in each row, we will print asterisk following discusse
d formula of 2 * row_number - 1
        printf("* "); // in 1st row, this loop will run 1 time as in 2nd time k will
be 1 (k was initialized earlier in outer for loop)
        ++k; // after k being 1, 2 * 1 -
1 will be 1 which is equal to k and execution won't enter while loop. Similarly in 2
nd row, i will be 2 and
    } // asterisk will be printed 2* 2 -
1 = 3, 3 times because in 4th time k will be equal to 3 and execution won't enter th
e while loop..this process will
    // go on by printing 1, 3, 5, 7, 9 11 etc asterisk row by row.
    printf("\n");
}
return 0;
}
```

Output:

Enter the number of rows: 5

```
 *
 ***
*****
*****
*****
```

Enter the number of rows: 8

```
 *
 ***
*****
*****
*****
*****
*****
*****
```

### Example 7: Full Pyramid of Numbers

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
```



## C Program

```
#include <stdio.h>
int main() {
    int i, space, rows, k = 0, count = 0, count1 = 0;
    //here we have to analyze the question, we have 1 number in 1st row, 3 in 2nd 5 in
    third..so it follows a sequence of 2n-1
    printf("Enter the number of rows: ");
    //suppose rows = 5
    scanf("%d", &rows);
    // first work is to print the spaces then the number '1'
    //and then on next line again spaces then numbers accordingly, as you can see no.
    of spaces to be printed depends on the row,
    // for the first row, spaces will be 5-1=4 second row will be 5-2=3 and so on.
    for (i = 1; i <= rows; ++i) {
        for (space = 1; space <= rows - i; ++space) {
            printf(" "); //printing spaces equal to 5-1=4 then next row 5-2=3 and so on
            ++count; // each time space is printed count is increased by 1,
            //first time it will be 4 because 4 spaces are printed.
        }
        while (k != 2 * i - 1) {
            //after printing spaces in each row,
            // we will print numbers following discussed formula of 2 * row_number - 1
            if (count <= rows -
1) { //as you can see numbers gets printed starting from row number
                // for second row it will start from 2 then 3 after that it prints in reverse
                order uptill row value i.e 2.
                // if the count is smaller or equal to row-
                1 then it will print in increasing order
                // and for printing in decreasing order this condition becomes false and control
                goes to else part.
                printf("%d ", i + k);
                ++count;
            } else {
                ++count1;
                printf("%d ", (i + k -
2 * count1)); //for printing the values in decreasing order uptill row number
            }
            ++k; // increasing the value of k by 1 each time the while loop executes,
            //as soon as the while condition becomes false, loop terminates
        }
        count1 = count = k = 0; // values of each assigned to zero for the next iteration
        printf("\n"); //for printing the next set of values in next line
    }
    return 0;
}
```

Output:

Enter the number of rows: 6

[Author: printf\("Electroverts Lab"\);](#)

```
1
2 3 2
3 4 5 4 3
4 5 6 7 6 5 4
5 6 7 8 9 8 7 6 5
6 7 8 9 10 11 10 9 8 7 6
```

## 32. C program to build a Basic Calculator.

In this example, you will learn to create a simple calculator in C programming using the switch statement.

To understand this example, we should have the knowledge of the following C programming topics:

[C switch Statement](#)

[C break and continue](#)

This program takes an arithmetic operator +, -, \*, / and two operands from the user. Then, it performs the calculation on the two operands depending upon the operator entered by the user.

```
#include <stdio.h>
int main() {
    char operator;
    double first, second;
    printf("Enter an operator (+, -, *, /): ");
    //entering the operation user wants to perform
    scanf("%c", &operator);
    printf("Enter two operands: ");
    //entering numbers
    scanf("%lf %lf", &first, &second);
    //we are using Switch case statement here because we have multiple conditions and
    // we need to perform different action based on the condition.
    // When we have multiple conditions and we need to execute a block of statements when
    a particular condition is satisfied.
    // In such case either we can use lengthy if..else-if statement or switch case.
    switch (operator) {
        // checks whether the operator is +, if yes then it executes this block,else go
        es to next.
        case '+':
            printf("%.1lf + %.1lf = %.1lf", first, second, first + second);
            //here,if the above condition is satisfied , and set of operation is done.
            // we use break to exit so that the control is not passed to other cases.
```

[Author: printf\("Electroverts Lab"\);](#)

```
        break;
        //checks whether the operator is -
, if yes then it executes this block,else goes to next.
        case '-':
            printf("%.11f - %.11f = %.11f", first, second, first - second);
            break;
        // checks whether the operator is *, if yes then it executes this block,else
goes to next.
        case '*':
            printf("%.11f * %.11f = %.11f", first, second, first * second);
            break;
        // checks whether the operator is /, if yes then it executes this block,else
goes to next.
        case '/':
            printf("%.11f / %.11f = %.11f", first, second, first / second);
            break;
        // operator doesn't match any case constant
        default:
            // if none of the case is satisfied,it will execute the default section.
            printf("Error! operator is not correct");
        }

        return 0;
}
```

Output:

Enter an operator (+, -, \*, /): +

Enter two operands: 10

22

10.0 + 22.0 = 32.0

Enter an operator (+, -, \*, /): /

Enter two operands: 44

11

44.0 / 11.0 = 4.0

## EXAMPLE PROGRAMS USING FUNCTIONS

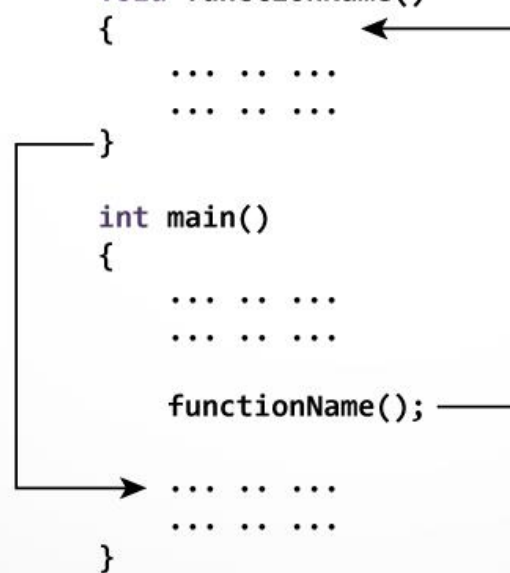
How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..
    functionName();
}

... ..
... ..
}
```



### 33. C program to sum each digit of a number

In this program, we will learn to count the sum of digits of a number entered by the user.

To understand this example, we should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C while Loop](#)

[C Functions](#)

```
#include<stdio.h>
void sum_digits(); //declaring the function
```

```
int main()
{
printf("Enter a number:"); //Get number by user
sum_digits(); //function call to calculate the sum
return 0;
}

void sum_digits()
{
    int n,sum=0,m;
    scanf("%d",&n); // storing the value inputted to variable n
    // suppose n=85
    while(n>0) //Repeat loop until number is greater than 0.
    {
        m=n%10; // Get the remainder of the number
        // we find the last digit using modulus operator , in this case m=5
        sum=sum+m; //sum the remainder of the number
        // sum=0+5=5 for the first time
        n=n/10; //dividing the number by 10 ,removes the last digit from n,
        //in this case n=8
        //now control goes to while loop it checks whether n>0, which is true,
        //enters the loop and the operation goes on till number is greater than zero.
    }
    printf("Sum = %d",sum); // final value of sum after while loop terminates gets printed
}
```

Output:

Enter a number:12345

Sum = 15

Enter a number:98759

Sum = 38

### [34. C Program to find the factorial of a number using user defined functions](#)

In this program we will learn to find the factorial of a number using user defined functions.

To have a better understanding of the program we have to have a good knowledge of the following concepts:

[C Programming Operators](#)

[C for Loop](#)

[C Functions](#)

[Author: printf\("Electroverts Lab"\);](#)

```
#include<stdio.h>

long factorial(int); // function declaration

void main()
{
    int number;
    long fact = 1; // set fact variable to 1

    printf("Enter a number to calculate it's factorial:\n"); //asking the user for a
number to calculate it's factorial
    scanf("%d", &number); //reading the number

    printf("%d! = %ld\n", number, factorial(number)); // printing the desired result
}

long factorial(int n) //function definition
{
    int c;
    long result = 1;

    for (c = 1; c <= n; c++) // for loop iterating over all number upto the number gi
ven by the user
        result = result * c; // calculate factorial

    return (result); // return the result
}
```

Output:

Enter a number to calculate it's factorial:

5

5! = 120

### [35. C program to check whether a number is palindrome or not using user defined functions](#)

In this program, we will find whether the number entered by the user is palindrome or not using user defined functions.

Before going through the code, the following concepts are prerequisites:

[C Programming Operators](#)

[C while Loop](#)

[C Functions](#)

[If statements](#)

[Palindrome condition](#)

[Author: printf\("Electroverts Lab"\);](#)

```
#include<stdio.h>
int checkPalindrome(int number) // function definition
{
    int temp, remainder, sum = 0; // initializing variables

    temp = number; // assigning variable number to a temporary variable

    while (number != 0) // while loop repeating until the number is not equal to zero
    {
        remainder = number % 10;
        sum = sum * 10 + remainder;
        number /= 10;
    }

    if (sum == temp) // checking if sum is equal to temp variable
        return 0;

    else
        return 1;
}

int main()
{
    int number;

    printf("Enter the number: "); // asking the user to enter a number
    scanf("%d", &number);

    if (checkPalindrome(number) == 0) // checking if the number is palindrome
        printf("%d is a palindrome number.\n", number);
    else
        printf("%d is not a palindrome number.\n", number);

    return 0;
}
```

Output:

Enter the number:1234  
1234 is not a palindrome number  
Enter the number:1221  
1221 is a palindrome number

## 36. C program to check if the number is prime or not using user defined functions

In this program let's learn to find whether the number entered by the user is prime or not using user defined functions.

Before that there are a few prerequisites that needs to be met:

[C Programming Operators](#)

[C for Loop](#)

[C Functions](#)

[If statements](#)

[Prime Number condition](#)

```
#include<stdio.h>
int checkPrime(int number) // function definition
{
    int count = 0; // setting count variable to 0

    for (int i = 2; i <= number / 2; i++) // for loop iterating over all numbers starting from 2 to the number upto half of the entered number
    {
        if (number%i == 0) // checking if the iter variable is divisible by 0
        {
            count = 1; // setting count variable to 1
            break; // breaking out of the for loop
        }
    }

    if (number == 1) // checking if number is equal to 1
        count = 1; // setting count variable to 1

    return count; // returning count variable
}

int main()
{
    int number;

    printf("Enter number: "); // asking the user for input
    scanf("%d", &number); // reading the input

    if (checkPrime(number) == 0) // checking if it is a prime number
        printf("%d is a prime number.", number);
    else
```



[Author: printf\("Electroverts Lab"\);](#)

```
        printf("%d is not a prime number.", number);

    return 0;
}
```

Output:

Enter number:120

120 is not a prime number

Enter number:291

291 is a prime number

## EXAMPLE PROGRAMS ON 1-D ARRAYS

### 37. C program to perform Simple Linear Search.

In this program, we will learn to find an element and display the location of it, within an array of numbers entered by the user.

To understand this example, you should have the knowledge of the following C programming topics:

[C Programming Operators](#)

[C for Loop](#)

[C if statements](#)

[Arrays](#)

[If statements](#)

```
#include <stdio.h>
int main()
{
    int array[100], search, c, n; // declaring the required variables

    printf("Enter number of elements in array:\n"); // asking the user for the length
of the array
    scanf("%d", &n); // reading the input

    printf("Enter %d integer(s)\n", n);

    for (c = 0; c < n; c++) // iterating over all numbers upto the entered length
        scanf("%d", &array[c]);

    printf("Enter a number to search\n"); // asking user to enter the element to be s
earched
    scanf("%d", &search);
    for (c = 0; c < n; c++) // iterating through the length of the array using for lo
op
```

[Author: printf\("Electroverts Lab"\);](#)

```
{
    if (array[c] == search)    /* If required element is found */
    {
        printf("%d is present at location %d.\n", search, c + 1);
        break;
    }
}
if (c == n)
    printf("%d isn't present in the array.\n", search);

return 0;
}
```

Output:

Enter number of elements in array

5

Enter 5 integer(s)

1

4

5

6

8

Enter a number to search

2

2 isn't present in the array.

Enter number of elements in array

6

Enter 6 integer(s)

1

2

3

4

5

6

Enter a number to search

3

3 is present at location 3.

## [38. C program to sort elements of an array using bubble sort](#)

In this program we will learn to sort an array in ascending order using the bubble sort method.

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

```
int main()
```

```
{
```

[Author: printf\("Electroverts Lab"\);](#)

```
int array[100], n, c, d, swap; // declaring all necessary variables

printf("Enter number of elements\n"); //reading the number of elements in the array
scanf("%d", &n);

printf("Enter %d integers\n", n);

for (c = 0; c < n; c++) // using a for loop to iterate through all elements of the array
    scanf("%d", &array[c]); // reading every array element

for (c = 0; c < n - 1; c++)
{
    for (d = 0; d < n - c - 1; d++)
    {
        if (array[d] > array[d + 1]) /* For decreasing order use < */
        {
            swap = array[d]; //bubble sort algorithm
            array[d] = array[d + 1];
            array[d + 1] = swap;
        }
    }
}

printf("Sorted list in ascending order:\n");

for (c = 0; c < n; c++)
    printf("%d\n", array[c]); // printing the sorted list of array

return 0;
}
```

Output:

Enter number of elements

5

Enter 5 integers

2

1

55

23

44

Sorted list in ascending order:

1

2

23

44

55

## EXAMPLE PROGRAMS ON STRINGS

### 39.C program to find the length of given string without using string functions

In this program we will learn to find the length of a given string without using in built functions. To understand this code well, we have to have knowledge of the following concepts of C programming:

[C operators](#)

[For loops](#)

[strings](#)

```
/* C Program to find the length of a String without
 * using any standard library function
 */
#include <stdio.h>
int main()
{
    /* Here we are taking a char array of size
     * 100 which means this array can hold a string
     * of 100 chars. You can change this as per requirement
     */
    char some_str[100], i;
    printf("Enter a string: \n");
    scanf("%s", &some_str);
    // gets();

    // '\0' represents end of String
    for (i = 0; some_str[i] != '\0'; ++i);
    printf("\nLength of input string: %d", i);

    return 0;
}
```

Output:

Enter a string:  
Programming

Length of input string: 11

Enter a string:  
Electrovert Labs

[Author: printf\("Electroverts Lab"\);](#)

Length of input string: 11 {Guess why length of Electrovert Labs is showing as 11 while we can count it as 16 taking count of space as 1 character}, that's because [scanf\(\)](#) stops taking input where it encounters space and for that we have another function called [gets\(\)](#) which takes space too.

## [40. C program to find the number of vowels, consonants and white spaces in a given string](#)

In this basic program we shall learn to count the number of vowels, consonants and white spaces in a string entered by the user and print it on the screen. Few concepts needed to understand the program are:

[C operators](#)

[For loops](#)

[If and else if conditions](#)

[Basics of strings](#)

```
#include <stdio.h>
int main() {
    char line[150];
    int vowels, consonant, digit, space;

    vowels = consonant = digit = space = 0; //initialising variables to 0

    printf("Enter a line of string: "); //asking the user for an input string
    fgets(line, sizeof(line), stdin); //reading the string using fgets

    for (int i = 0; line[i] != '\0'; ++i) {
        if (line[i] == 'a' || line[i] == 'e' || line[i] == 'i' ||
            line[i] == 'o' || line[i] == 'u' || line[i] == 'A' ||
            line[i] == 'E' || line[i] == 'I' || line[i] == 'O' ||
            line[i] == 'U') {
            ++vowels;
        } // checking for uppercase and lowercase vowels in a string and updating the
        vowels variable
        else if ((line[i] >= 'a' && line[i] <= 'z') || (line[i] >= 'A' && line[i] <=
        'Z')) {
            ++consonant;
        }
        else if (line[i] >= '0' && line[i] <= '9') {
            ++digit; // checking if the string has any digit
        }
        else if (line[i] == ' ') {
            ++space; //checking if the string contains a space
        }
    }
```

```
}  
  
printf("Vowels: %d", vowels);  
printf("\nConsonants: %d", consonant);  
printf("\nDigits: %d", digit);  
printf("\nWhite spaces: %d", space);  
return 0;}
```

Output:

Enter a line of string: Programming is fun 23

Vowels: 5

Consonants: 11

Digits: 2

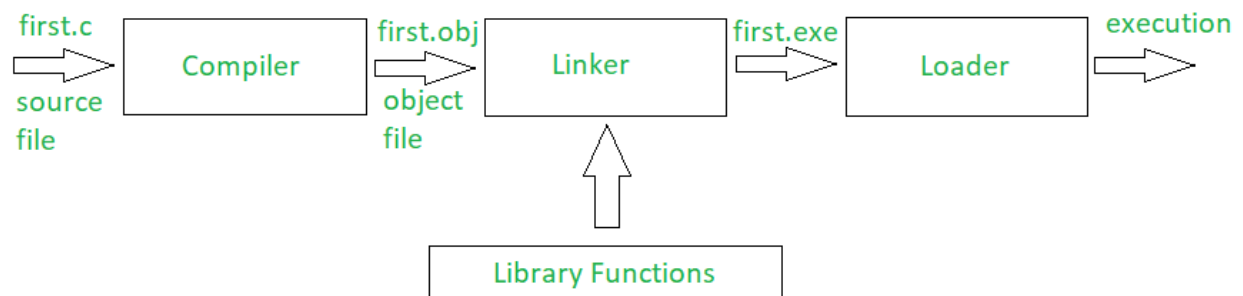
White spaces: 3

## Additional Content

### 1. How does a C program executes?

Whenever a C program file is compiled and executed, the compiler generates some files with the same name as that of the C program file but with different extensions. So, what are these files and how are they created?

Below image shows the compilation process with the files created at each step of the compilation process:



Every file that contains a C program must be saved with '.c' extension. This is necessary for the compiler to understand that this is a C program file. Suppose a program file is named, first.c. The file first.c is called the source file which keeps the code of the program. Now, when we compile the file, the C compiler looks for errors. If the C compiler reports no error, then it stores the file as a .obj file of the same name, called the object file. So, here it will create the first.obj. This .obj file is not executable. The process is continued by the Linker which finally gives a .exe file which is executable.

**Linker:** First of all, let us know that library functions are not a part of any C program but of the C software. Thus, the compiler doesn't know the operation of any function, whether it be printf or scanf. The definitions of these functions are stored in their respective library which the compiler should be able to link. This is what the Linker does. So, when we write #include, it includes stdio.h library which gives access to Standard Input and Output. The linker links the object files to the library functions and the program becomes a .exe file. Here, first.exe will be created which is in an executable format.

**Loader:** Whenever we give the command to execute a particular program, the loader comes into work. The loader will load the .exe file in RAM and inform the CPU with the starting point of the address where this program is loaded.

**Instruction Register:** It holds the current instructions to be executed by the CPU.

**Program Counter:** It contains the address of the next instructions to be executed by the CPU.

**Accumulator:** It stores the information related to calculations.

The loader informs Program Counter about the first instruction and initiates the execution. Then onwards, Program Counter handles the task.

#### Registers in CPU

Instructions Register

Program Counter

Accumulator

#### Difference between Linker and Loader

LINKER	LOADER
Linker generates the executable module of a source program.	Loader loads the executable module to the main memory for execution.
Linker takes the object code generated by an assembler, as input.	Loader takes executable module generated by a linker as input.
Linker combines all the object modules of a source code to generate an executable module.	Loader allocates the addresses to an executable module in main memory for execution.
The types of Linker are Linkage Editor, Dynamic linker.	The types of Loader are Absolute loading, Relocatable loading and Dynamic Run-time

	loading.
--	----------

## 2. When you are installing a program, are you really just compiling it?

Typically, if you are a standard front-end end-user your programs are already compiled for your architecture and OS/Kernel. You click an executable file and it runs. Mean your average "install.exe" or "setup.exe" tends to produce a binary executable in the install directory (this may or may not come with a bunch of sub-folders, resource files, config files, maybe other bytecode files and scripts etc. as well).

The progress bar is doing something...

With installers, while the installation progress bar and accompanying messages appear to be generating a bunch of files, this isn't compilation either.

What is usually going on here decompression (unzipping), copying and writing text/ short code/ hexcode etc. type files and verifying integrity of files.

There are exceptions of course

Although some dynamic generation and calculation can occur from an installer - this is typically not the same as compiling. Though that's not to say some installers don't run a form of compiling for some architecture specific alterations during install... but this small compiling process doesn't make the whole installer a compiling process.

Where compiling traditionally happens

Compilation is usually actioned by a developer/programmer/engineer - this essentially makes it something they can ship to users.

Example of where compilation and install can be the same process... kind of... ok maybe not e.g. If you're running a flavor of Linux, and as an end-user... the repository/app-store/etc. doesn't have the software, or the exact version of software you want, you may end up downloading the source code and compiling it yourself in order to then install it for your system. While installation and compiling are still really different things in this process, there are tools to compile the source and install the binaries all in one go, making the demarcation between the processes blurry if you don't know what is really going on.

## 3. How is a programming language created and developed?

Programming languages will give you a good idea of where and how computers started and progressed. If you want to get into the history of computers I'd suggest you to look at things such



as the Jacquard loom and Babbage's difference engine. These two inventions probably had the greatest early influence on modern computing. Usage of punched cards (Jacquard) and the mechanical 'calculation' (Babbage) provided a great foundation.

So let's start with how the computer works. At the heart of every computer is a Transistor (and before that vacuum tubes). The interesting thing behind a transistor is it can send electricity in two different ways, depending on its state. This allows for the creation of logical flows of electricity. With this we can create all sorts of wonderful things: NAND gates, AND gates, half adders, multiplexes etc.

Now that we have these electronic building blocks in place, there are essentially two types of signals that come into them. Signals that tell them what to do, and actual data that they should compute with. So the command to add, might (say) take the data from register 1, add it to the data in register 2, and store that information in register 3. What this command does is sets the computer up in a state so that registers 1 and 2 behave as inputs to the adder, and register 3 stores the result. The same would be true for subtraction multiplication etc. There are also commands to say jump to a certain line, read information from memory etc.

The binary commands these binary commands are the machine code necessary to 'set the CPU.

So now we have a computer that runs on machine code. Nothing more nothing less. Now this is a very hard to use, machine. So assembly is almost always one of the first languages that's created. To use assembly we need to create an assembler. An assembler is essentially a compiler that turns assembly language into machine code. As a result assembly languages are 1 to 1 with the machine code commands. The idea being since we're coding this in binary, it's a good idea to keep it simple. So now we have something that can turn assembly language into machine code.

So now we have two levels of 'languages' 0 - Machine code: This is code that the CPU understands, it's in binary, and not very user friendly 1 - Assembly language: This uses some 'English like' terms, but is still relatively clunky, and 1 to 1 command wise with the machine code.

So let's add a third,

2- High level language.

A high level language is something that more closely resembles English, such as C. We have loops and data structures and other tools. To use C we have to write a compiler. A compiler takes code written in the C language and creates object code (similar to assembly language). Then another program turns that object code into machine language. Anymore these two steps are usually combined into one for efficiency sake. Now we have your first definition. A compiler turns a high level language into object (or machine) code.

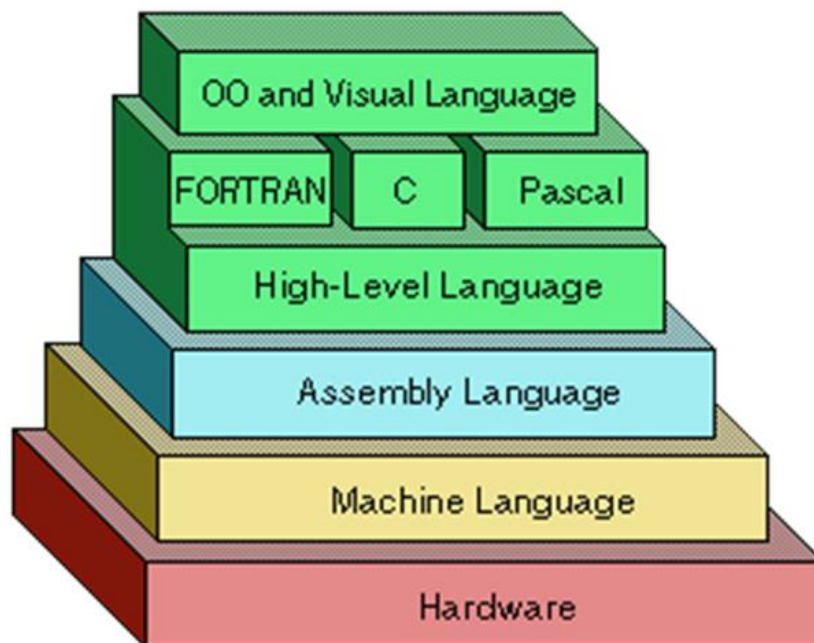
[Author: printf\("Electroverts Lab"\);](#)

Now that we have our first high level language (C) it probably seems foolish and painful to work with the previous assembly language unless we have to. So now we can write new languages and compilers in C, or any other language we've now made.

So now let's tear into an interpreter. An interpreter is a program that reads, and executes a program on its own. Instead of turning a program that turns high level code into machine code, this reads the high level code, (usually a line at a time) and executes it.

Let's take java for example. Java is an interpreted language, basically that means someone has made a program (in say C) This C program, reads the java and executes it. So there's another layer between the computer and the code.

There's a lot of stuff this response kind of glazes over, how to build a CPU, the considerations into making a language, how the same language can run on different CPUs. The advantages of interpreters over compilers etc. But hopefully it gives a good amount of background and information that you can read up and research on your own.



#### [4. Why Are There So Many Programming Languages?](#)

The short answer is that we don't need so many languages, but we want them. Let's explore this further.

Aren't they all the same?

In a sense, yes. You can create a web site using Ruby, Java, Python, C#, Go or JavaScript. You can use C or C++ or Haskell or Rust. Or COBOL or Pascal or Perl.

Underlying this fact is that all of these languages serve the same purpose: to turn human thoughts into the 1's and 0's that the computer understands. In highfalutin computer terms, they are all "Turing complete".

At their most foundational level, these languages are all the same. But on the surface – where humans interact with them – they vary a lot. This is where other concerns come into play.

#### Different tools for different jobs

Programming languages are tools, and we choose different tools for different jobs. A tractor trailer and a bicycle and a Tesla are all vehicles – they have wheels and steering and will get you from point A to point B – but obviously we use them for different things.

Programming languages are similar. Ruby and JavaScript are great for building web sites; Java and C++ are often used for financial trading; Python and R are the tools of choice for analyzing statistics.

Languages often make trade-offs in terms of convenience, safety, and speed – much like vehicles. The trade-off is dictated by the job at hand.

#### Developers have tastes

Beyond mere utility, developers choose tools based on personal tastes.

A programming language is a tool for humans to express ideas to computers. While we developers have many things in common, there is natural variety in the way our minds work.

Because we have many choices of good programming languages, we can select one that "works the way I think". Some developers like Ruby's flexibility, while others prefer Java's strictness. Where some languages feel like math, others look like prose.

#### People first

Beyond utility, and beyond taste, businesses run on people. Often, you will choose a programming language based on what you, or the people around you, know.

Technologies are supported by "ecosystems" – communities and organizations that provide the tools and assistance that every developer needs. A good ecosystem – Ruby has a great one, for example – can make the individual developer more successful.

#### Variety is strength

In summary, we have a variety of programming languages because there is a variety of jobs to be done and a variety of people who do those jobs. This diversity makes interesting programs – and interesting companies, and interesting careers – possible.

## 5. History of C Language

History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the founder of the c language.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.

Let's see the programming languages that were developed before C language.

Language	Year	Developed By
<b>Algol</b>	1960	International Group
<b>BCPL</b>	1967	Martin Richard
<b>B</b>	1970	Ken Thompson
<b>Traditional C</b>	1972	Dennis Ritchie
<b>K &amp; R C</b>	1978	Kernighan & Dennis Ritchie
<b>ANSI C</b>	1989	ANSI Committee
<b>ANSI/ISO C</b>	1990	ISO Committee
<b>C99</b>	1999	Standardization Committee