An algorithm can be defined as “a sequence of steps to be followed for getting the desired output for a given input.”  
  
The sequence of steps of an algorithm can be stated in a human readable language (English) or in [pseudo-code](https://en.wikipedia.org/wiki/Pseudocode) (i.e, - a notation resembling a simplified programming language).  
  
After the **analysis** stage of **SDLC**, producing an **algorithm** is the first step towards solving the given problem.  
  
There may be several ways to solve a given problem, hence there may be **several algorithms for a given problem**.  
  
Before attempting to write an **algorithm**, one should identify the expected inputs and outputs for the given problem.  
  
An algorithm has the following properties:

1. Finiteness: An algorithm must always terminate after a finite number of steps.
2. Definiteness: Each step of an algorithm must be precisely defined, i.e., the step should perform a clearly defined task without much complication.
3. Input and Output values: The steps of the algorithm are designed to work on the input values and then produce the desired output in the final step.
4. Effectiveness: The efficiency of the steps and the accuracy of the output determine the effectiveness of the algorithm.

When studying algorithms, it's important to know the different problem types to understand the diverse techniques and approaches used to solve them. Here is a classification of important problem types in algorithms:

* **Sorting and Searching**

1. **Sorting Algorithms:**Fundamental for organizing data efficiently.
2. Examples: Bubble Sort, Merge Sort, Quick Sort, Heap Sort.
3. **Searching Algorithms:** Key for efficient data retrieval.
4. Examples: Linear Search, Binary Search.

* **Divide and Conquer**

1. **Technique:** Breaks down complex problems into simpler subproblems, a strategy that is widely applicable.
2. **Examples:** Merge Sort, Quick Sort, Binary Search.

* **Dynamic Programming**

1. **Technique:** Optimizes by storing results of overlapping subproblems, crucial for solving complex problems efficiently.
2. **Examples:** Fibonacci Sequence, Knapsack Problem, Longest Common Subsequence.

* **Greedy Algorithms**

1. **Technique:** Builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit.
2. **Examples:** Huffman Coding, Dijkstra’s Algorithm, Prim’s Algorithm.

* **Graph Algorithms**

1. **Applications:** Vital for solving problems related to networks, paths, and connectivity.
2. **Examples:** DFS, BFS, Dijkstra’s Algorithm, Minimum Spanning Tree Algorithms.

* **Backtracking**

1. **Technique:** Systematically searches for a solution to a problem by trying out different possibilities.
2. **Examples:** N-Queens Problem, Sudoku Solver.

* **String Algorithms**

1. **Applications:** Important for text processing and manipulation.
2. **Examples:** KMP Algorithm, Rabin-Karp Algorithm.

* **Computational Geometry**

1. **Applications:** Useful in graphics, CAD, and geographic information systems.
2. **Examples:** Convex Hull, Closest Pair of Points.

* **Number Theory and Cryptography**

1. **Applications:** Foundational for security and encryption algorithms.
2. **Examples:** Euclidean Algorithm, RSA Algorithm.

* **Optimization Problems**

1. **Applications:** Central in finding the best solution among many possibilities.
2. **Examples:** Linear Programming, Network Flow Problems.

**Why These Types are Essential**

1. **Foundation in Theory and Practice:** Understanding these problem types provides a foundation in both the theoretical underpinnings and practical applications of algorithms.
2. **Broad Applicability:** These problems and techniques are widely applicable across different domains, from data processing and analysis to networking and cryptography.
3. **Algorithm Design Patterns:** Learning these problem types helps in recognizing and applying common algorithm design patterns, such as divide and conquer, dynamic programming, and greedy strategies.
4. **Problem-Solving Skills:** Tackling a variety of problem types enhances problem-solving skills, critical thinking, and the ability to design efficient algorithms for new problems.

Until now, we have seen several examples of simple algorithms in which a formula is applied to the given input to arrive at the desired output.  
  
However, in real life, algorithms are not so simple. Most of the algorithms employ conditional branching into different flows (or sequence of steps) depending on the input or an intermediary value.  
  
If the condition is true, one of the two branches is explored; else, the other alternative is taken.  
  
This is usually accomplished using the if-then-else construct in algorithms. This structure is also known as the **selection structure**,i.e.,, the selection is based on the result of the condition which can either be **true** or **false**.  
  
Given below is a sample **algorithm** which uses branching for finding the **largest of two given numbers**:

Step-1: Start

Step-2: Read number1 and number2.

Step-3: if number1 **>** number2 then largest ⟸ number1.

Step-4: if number2 **>** number1 then largest ⟸ number2.

Step-5: Display the largest value.

Given below is a sample **algorithm** which uses branching to identify if the **given number is even or odd**:

Step-1: Start

Step-2: Read a number.

Step-3: Compute remainder ⟸ number % 2.

Step-3: if (remainder = 0) then // here remainder = 0 means comparing whether remainder contains value 0 or not

print "number is Even".

else

print "number is Odd".

What is the purpose of pseudocode in algorithm development?

To provide a high-level description of an algorithm using natural language.

Consider the following example for developing an algorithm, flowchart and the corresponding program in **C** to **find the largest of three numbers**.  
  
The following data is gathered after the analysis of the above given problem statement:

Problem Input: three integer numbers

Problem Output: largest of the three given integer numbers

Relevant logic: Comparison among the three numbers using > operator.

Given below is a sample **algorithm** for the given problem:

Step-1: Start

Step-2: Read 3 integer numbers a, b, c.

Step-3: if (a > b) && (a > c) then

print "a is the largest number".

else if (b > c) then

print "b is the largest number".

else

print "c is the largest number".

Step-4: Stop

Until now we have learnt about algorithms which use a simple sequence of steps and also employ branching using the if-then-else construct.  
  
There may arise a situation in an algorithm where, we might have to repeat a single step or a set of steps multiple times until a condition is satisfied to arrive at the desired result.  
  
This repetition of steps is referred to as Iteration and is accomplished using a construct called Loops.  
  
It is represented by the while, do-while and for constructs in most programming languages.  
  
Given below is a sample **algorithm** that demonstrates the concept of looping to **print even numbers between 2 and 99**.

Step-1: Start

Step-2: Initialize num ⟸ 2.

Step-3: Display the value of num.

Step-4: Increment the value of num by 2, i.e, num ⟸ num + 2.

Step-5: if (num < 99) then

go to Step-3 //this is the step which repeats the steps from Step-3

Step-6: Stop

While looping, we must ensure that there is a condition which **terminates** the loop after some **finite number of iterations**. Otherwise it ends up as an **infinite loop** (a common mistake while writing loops).

A flowchart is a **pictorial representation** of the sequence of steps which solve a given problem.  
  
Flowchart can also act as a **blueprint** for solving a problem.  
  
Some of the main advantages of a **flowchart** are:

1. It provides an easy way of communication since a non-programmer also can easily understand the steps indicated.
2. It clearly depicts the sequence of steps which detail the operations performed.
3. It helps us in verifying the logic of the proposed solution even before we start coding.
4. It provides a clear overview of the entire program with its solution.

**Unit 1 - Lesson 2 - Comment Lines, Identifiers, Variables, Keywords**

**2.1.1. Understanding comments in C**

As mentioned earlier, a computer program is a collection of instructions or statements.  
  
A **C** program usually consists of multiple statements.  
Each statement is composed of one or more of the **three** given below:

1. Comments
2. Whitespace characters
3. Tokens

In a computer program, a comment is used to mark a section of code as non-executable.  
  
Comments are mainly used for two purposes:

1. To mark a section of executable code as non-executable, so that the compiler ignores it during compilation.
2. To provide remarks or an explanation on the working of the given section of code in plain English, so that a fellow programmer can read and understand the code.

In **C**, there are two types of comments:

1. **end-of-line comment** : It starts with //. The content that follows the // and continues till the end of that line is a comment. It is also called as **single-line comment**
2. **traditional comment** : It starts with /\* and ends with \*/. The content between /\* and \*/ is the comment. It is also called as **multi-line comment**.

The code given below shows the two types of comments:

/\*

C programming language was developed by Dennis Ritchie.

This is called a header comment which is used to describe

what this program would do. As you can notice the comment is

spanning across multiple lines.

\*/

#include <**stdio.h**>

**void** main() {

**int** num1 = 10, num2 = 20;

printf("sum of two numbers = %d", num1 + num2);

}//end of the main() function - this is an example of a end-of-line comment

**Given below are 3 important points regarding comments:**

1. There **should not** be any space between the two forward slashes in //, i.e., / / is incorrect. Similarly, there should not be any space between the **slash** and **star** characters in /\* and \*/, i.e., / \* and \* / are incorrect.
2. **Comments do not nest**, i.e., /\* and \*/ comment has no special meaning inside a // comment.Similarly, a // comment has no special meaning inside a /\* comment.
3. One should not write comments inside [character literals](https://en.wikipedia.org/wiki/Character_literal) (i.e., characters enclosed between single-quotes). Comments inside [String literals](https://en.wikipedia.org/wiki/String_literal) (i.e., text enclosed between double-quotes) are treated as part of the String's content.

**2.1.3. Whitespace characters and Tokens in C**

**Whitespace Characters**  
  
In English,we use space to separate two words. When it comes to typing text in a computer, there are different types of characters that are used to separate text by creating a space. These are called [whitespace](https://en.wikipedia.org/wiki/Whitespace_character) characters.  
  
The different **whitespace** characters in **C** are:

1. Space ' ' – (ASCII SP) produced by pressing **spacebar**
2. Tab '\t' – (ASCII HT) produced by pressing the **tab key**
3. [Form Feed character](https://en.wikipedia.org/wiki/Page_break#Form_feed) '\f' – (ASCII FF) usually used as the page separator char between lines/paragraphs
4. **Line Terminator chars** (used to separate two lines) – produced by pressing **Enter** key:
   1. [Line Feed](https://en.wikipedia.org/wiki/Newline) - '\n' (ASCII LF also called NL - New Line) - used in all Unix and Mac OS X systems
   2. [Carriage Return](https://en.wikipedia.org/wiki/Carriage_return) - '\r' (ASCII CR) – used in MAC OS 9 and below
   3. Carriage Return followed by Line Feed- '\r\n' (ASCII CRLF) – used in Windows systems

**Tokens**  
  
The basic buildings blocks used to write a C program are called **tokens**.  
  
Consider the example given below:

#include <**stdio.h**>

**void** main() {

**int** num1 = 10, num2 = 20;

printf("sum of two numbers = %d", num1 + num2);

}

Individual fragments like **void**, **main**, **{** etc are different types of tokens.  
  
In **C**, tokens can be classified into **six** categories as given below :

1. **Identifiers** – These are simple names used to refer to or identify something. For example, names of variables, functions are called **Identifiers**. In the code given above, **main**, **num1**, **num2**, **printf** are Identifiers.
2. **Keywords** – These are one of the [32 reserved words](https://en.wikipedia.org/wiki/C_(programming_language)#Reserved_words) like int, for, if, etc. These words have special meaning when used as part of the program. We shall learn more about them in later sections.
3. **Constants** – These are fixed values like 10, 20 etc, which are used in a program.
4. **String constants** – String constants are specified within double quotations. Example-"Total", "CodeTantra" etc.
5. **Separators** – The following are called as **separators** - ( ) { } [] ; , . .
6. **Operators** – The following are called **operators**, - = > < == >= etc. We shall learn more about them in later sections.

**2.1.4. Understanding escape sequences in character and string literals**

In **C**, the backslash character \ is used to mark an [escape sequence](https://en.wikipedia.org/wiki/Escape_sequences_in_C). An **Escape Sequence** is an escape character \ followed by a normal character. For example: \n or \t.  
  
The presence of the escape character changes the meaning of the character which follows it. For example, when the string literal "Hello\tWorld" is printed, the result is seen as

Hello World

In the string literal "Hello**\t**World", \t represents the **TAB** character.  
  
Similarly, if we want to print a **double quote** inside a double-quoted string literal, we need to escape the **double quote** by using the escape character \. For example :

printf("Hello \" (Quote)");

**Variables & Keywords**

**2.2.1. Character set of C language**

Each and every language in the world requires alphabets to form words. Likewise, a programming language also needs a **set of characters** to write a program.  
  
The set of characters used in a language is known as its **Character Set**.

Every language makes use of its own character set to form words or symbols that make up the **vocabulary** of the language.  
  
**C** language is [case sensitive](https://en.wikipedia.org/wiki/Case_sensitivity). By case sensitive, we mean that the C compiler treats lowercase and uppercase characters differently. For example, the variable name **num1** is different from **Num1**.  
  
**C** language has its own character set. The character set for [ANSI Standard C](https://en.wikipedia.org/wiki/C_(programming_language)#Character_set) (ANSI C) is as follows:

Uppercase alphabets: **A** to **Z**

Lowercase alphabets: **a** to **z**

Decimal digits: **0** to **9**

Special characters: **+ - \* / % = < > : ; , .’ “ ?! # \ ( ) { } \_ [ ] & | ^ ~**

[Escape sequences](https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences): **\b \t \v \r \f \n \\ \' \"\? \0 \a**

**2.2.2. Understanding Identifiers**

In any language, we learn **words** after learning the alphabet. We use words to name and identify different things. These names are nothing but **Identifiers**.  
  
**Identifiers** are **names** used to refer to any entity in a program. (A program can contain many entities (or building blocks) such as data types, constants, variables, functions, arrays, etc. We shall learn about them in the later sections.)  
  
An **identifier** is a sequence of characters. In **C**, **identifiers** can be formed by combining **alphabets**, **digits** and a special character **underscore** '**\_**' .  
  
For example, consider the program given below:

#include <**stdio.h**>

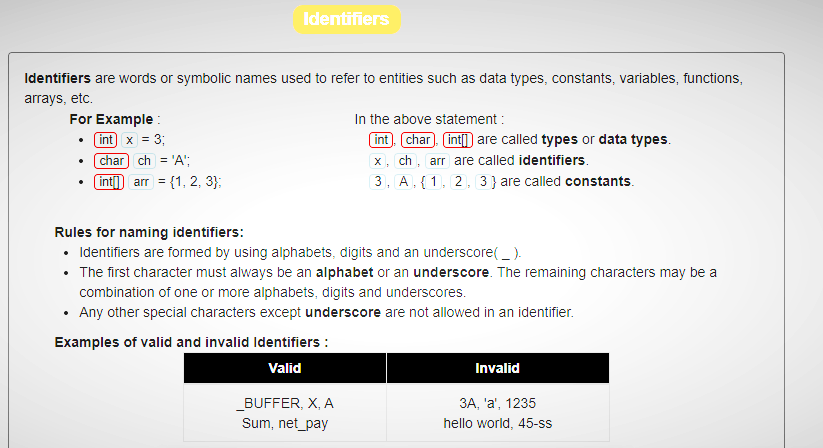
**void** main() {

printf("**Hello!**");

}

The tokens **main** and **printf** are the names of two functions and are called **identifiers**.  
  
Given below are the rules for creating a valid **identifier** in **C**:

* The first character must always be an **alphabet** or an **underscore**. The remaining characters can be a combination of one or more alphabet, digits, and underscores.
* No special characters except the **underscore** are allowed in an identifier.
* An identifier can be of any length. However, in old **C** (before C was standardized by ANSI), only the first **8** characters were considered by the compilers when the names were compared for equality. This limit was later changed to **31** characters as per the ANSI standard.



**2.2.4. Understanding Variables**

A **variable** is a name given to a memory location to **store** some value. Since the memory location can store **different values** during execution of a program, the name used to refer to it, is called a **variable**.

Since **variables** are a part of **identifiers**, they follow the same naming conventions. Like **identifiers**, a valid **variable** name can start with an alphabet or an **underscore** ( **\_** ) and later have a combination of one or more letters, digits and underscores.  
  
A few examples of valid variable names are : sum,total,average\_marks, etc.  
  
When creating a **variable**, we should mention the type of data (for example, integer or character) that it would store. This is called the **data type** of that **variable**.  
  
In a **C** program, variables should be **declared** before their usage.  
  
The format for **declaring a variable** is data\_type **variable\_name**;.  
  
For example:

**int** count; // int is the data type and count is the variable name

The above declaration can also be combined with initialisation. In such a case, the format for **declaring a variable** is data\_type **variable\_name** = constant\_value;.  
  
We shall learn more about **data types** and **constants** in the later sections.  
  
Given below is an example of declaring and initialising a variable in the same line:

**int** count = 9; // Here the variable count is being initialised to a constant value 9

#include <**stdio.h**>

**void** main() {

**int** age = 34;

printf("age : %d", age); **}**

**2.2.5. Understanding Keywords**

There are certain **predefined words** as part of **C** programming language that have a special meaning and purpose. They are called **reserved words** or **keywords**

The user (programmer) cannot redefine these keywords, i.e.,the user cannot change their spellings or add new ones to the **C** programming language.  
  
All reserved words in **C** are formed using only the **lowercase** letters.  
  
Keywords have special meaning and purpose, so they cannot be used as **variable** names.  
  
**C** language has **32** reserved words as per ANSI standards. They are as given below:

auto break case char

const continue default do

double else enum extern

float for goto if

int long register return

short signed sizeof static

struct switch typedef union

unsigned void volatile while

**Unit 1 - Lesson 3 - Syntax of main() and printf() with exercise**

3.1.1. Introduction to main() function

In **C** programming language, execution of the code starts with a [function](https://en.wikipedia.org/wiki/Subroutine) called main

We shall learn more about functions in the later sections. For now, we can safely assume that **function** is the name given to a set of one or more executable statements. main() is a **user defined function**, i.e., a user (a programmer) writes the code for the main() function.  
  
While executing a **C** program, the **Operating System (OS)** only calls the main() function in that program.  
  
When the **OS** executes a program, the program usually returns an integer value 0 if the execution of that program is successful.  
  
In **C**, **main()** can be written in such a way that it either returns a void or an int.  
**#include <stdio.h>**

**void main() {**

printf("Sample main() function with void as return type!");

**}**

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

printf("Sample main() function with int as return type!");

return 0;// 0 value indicates that the execution is successful  
**}**

If the programmer does not specify any return type, the return type is by default considered as int.  
  
The return type of main() can also be void, which means that it does not return anything. We shall learn more about **void** and **return type** is in the ensuing sections.  
  
The name of the main() function should always be in lowercase, i.e., if a function is written as Main(), it is not the main function which is called by the **OS**.

**3.1.3. Usage of Spaces and Tabs**

**Spaces** and **Tabs** are very important while writing code in any programming language. Appropriate usage of spaces and tabs improves the readability of the given code. Such code is called a formatted code.  
  
An **empty space** is used as a separator between two tokens. A **tab** **———**  is used only to **indent lines**.

Statements enclosed between an **opening brace** { and a **closing brace** } are referred to, as a **block of code**.  
  
Whenever a new block of code starts after the **opening brace** {, the lines inside that block should be**indented by one level** (one level means by one **Tab**) compared to the line that contains the opening brace {.

**Note:** Usually after an opening brace {, we press **ENTER** **¬**  key to go to the next line, where a **tab** is automatically inserted for indentation.  
  
The example given below shows the usage of spaces and tabs:

**#include <stdio.h>**  
void main() {

**int** i;

**for** (i = 0; i < 5; i++) {

printf("i = %d\n", i);

}

}

orange borders mark spaces

——— pink borders mark tabs

**Syntax of printf() function**

**3.2.1. Understanding how to display output using printf() function**

The line printf("Hi, I am learning C programming!\n"); is responsible for printing the text.

This line is written inside the main( ) function. Executable statements must be placed inside such function definitions. We shall learn more about functions in the coming sections.  
  
In **C**, **string constants/literals** (i.e. text) must be enclosed in **double quotes**.  
  
In the above given example, the string literal Hi, I am learning C programming!\n is enclosed in **double quotes**.

The code given below demonstrates the usage of printf() function to print **integers**, **floats**, etc,. using format specifiers.

#include <**stdio.h**>

void main() {

printf(**"sum of integer numbers = %d\n"**, 10 + 20);

printf(**"sum of float numbers = %f\n"**, 10.56 + 20.49);

}

The format specifiers %d and %f are used in printf( ) statement to print the **integer** and **float** values from their respective expressions.

**Unit 1 - Lesson 4 - Basics of Bits and Bytes - Binary, Decimal, Octal and Hexa-decimal systems**

4.1.1. Understanding Number System

In computers we normally use four different numbering systems - Decimal, Binary, Octal and Hexadecimal.  
  
The decimal system is a number system which is used in our day-to-day transactions like currency,counting etc. In this system,the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are used to denote various numbers.  
  
In Binary number system, 0's (zeros) and 1's (ones) are the only symbols that are used to represent numbers of all magnitudes (sizes). For example, a normal decimal number 3 (three) is represented in a binary as 11. We shall learn more about it in the later sections.  
  
Binary system is mostly used in computers and other computing devices.  
  
A number in a particular base is written as (Number)base of number. For example (34)10 is a decimal number (Thirty Four) and (11)2 is a binary number 11 (it is read as One One and not as Eleven) which actually represents a decimal number whose value is 3.  
  
Since the decimal number system is more commonly used,the decimal number (124)10 is simply written as 124. However, if the same number has to be represented in binary system, it is written as (1111100)2.  
  
Similarly, the octal number system uses 8 as its base. It is generally used in digital displays and in representing file permissions under UNIX/Linux operating systems.  
  
Hexadecimal system or Hex is a number system that uses 16 as the base to represent numbers. We shall learn more about octal and hexadecimal number systems in the later sections.