UNIT – 1

Digital Computer

A computer is an electronic data processing device, which accepts and stores data input in binary format, processes this binary data input, and generates the output in a required format. A computer is a combination of hardware and software resources which integrate together and provides various functionalities to the user.

Hardware represents the physical and tangible components of a computer, i.e. the components that can be seen and touched.

Examples of hardware

- Input devices − keyboard, mouse
- Output devices − printer, monitor
- Secondary storage devices − Hard disk, CD, DVD
- Internal components − CPU, motherboard, RAM

Software is the set of programs or instructions that are required by the hardware resources to function properly.

Examples of software

- Operating System
- Compiler

X

- Loader

Functional Components of a Digital Computer

Working of a computer includes following three major tasks:

1. Data Input
2. Processing Input
3. Data Output

The basic components that help in performing above mentioned tasks are called as the functional components of a computer.

These functional components are:

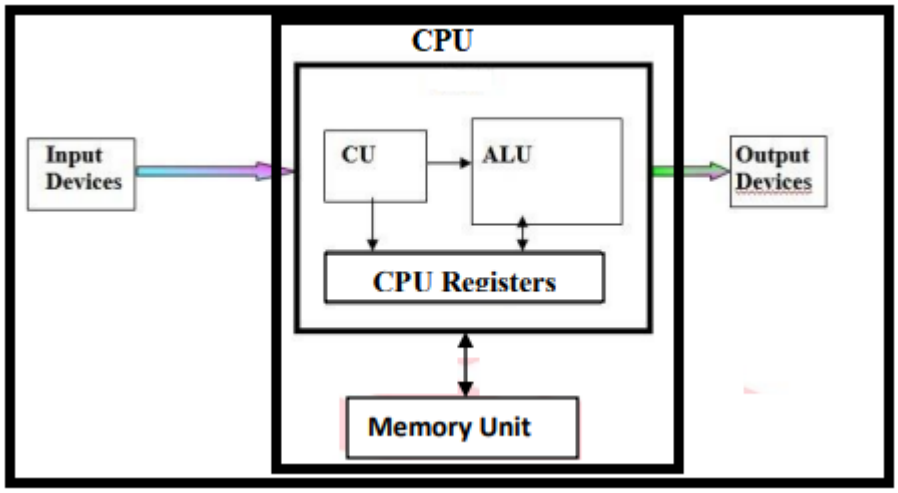
Input unit: It takes the input from input devices

Central processing unit: It does the processing of data

Output unit: It produces the output and helps in visualizing it.

Memory unit: It holds the data and instructions during the processing of data.

Block Diagram of a Digital Computer



Input Unit: The input unit consists of input devices that are attached to the computer. These devices take input and convert it into binary language that the computer understands. Example: keyboard, mouse, joystick, scanner, etc.

Central Processing Unit (CPU):

The CPU is called the brain of the computer as it controls the working of computer. CPU processes the information input by the input devices. CPU first fetches the instructions from memory and then interprets them so as to know what is to be done. It performs all required computation. After computation it stores the output or displays it on some output device.

The CPU consists of three main components

i. Arithmetic Logic Unit (ALU):

The ALU performs arithmetic and logical operations. Arithmetic calculations include addition, subtraction, multiplication and division. Logical calculation involves relational and logical operations.

ii. Control Unit (CU)

The CU coordinates and controls flow of data and instructions among various units of a computer. It also controls all the operations of ALU, memory registers and input/output units. It takes care of executing all the instructions stored in a program by fetching the instructions, decoding the instructions and sending the appropriate control signals to other functional units until the required operation is completed

iii. CPU registers

A register can hold the data, instruction, and address of memory location, which is to be directly used by the processor. Registers can be of different sizes (16 bit, 32 bit, 64 bit and so on).

There are two major types of registers:

User/ General Purpose registers: It is used to store operands, intermediate results, etc. during assembly language programming.

Accumulator (ACC): It is the main register and contains one of the operands of an operation to be performed in the ALU.

Output Unit: It is composed of output devices attached to the computer. It converts the binary output data coming from CPU to human understandable form. Some examples of different output devices are monitor, printer, plotter, etc.

Storage Devices

Storage device is any hardware capable of holding information either temporarily or permanently.

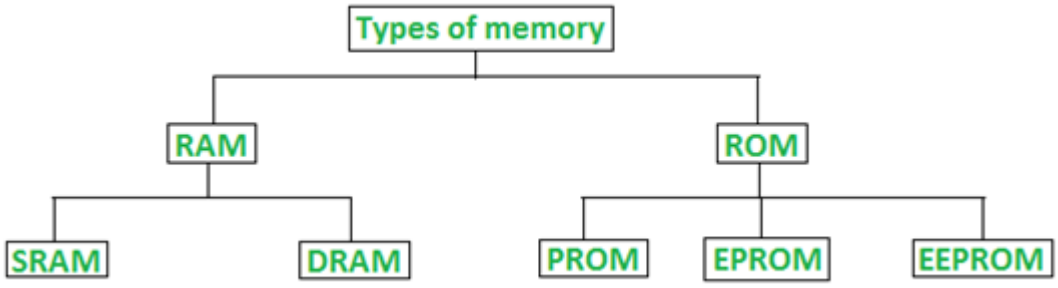
There are two types of storage devices used with computers:

- **Primary storage device** or Primary Memory such as Random Access Memory (RAM) and Read Only Memory (ROM)
- **Secondary storage device** or Secondary memory such as hard drive, CD, DVD, magnetic tapes.

Memory Unit

Computer memory is of two basic types: Primary memory (RAM and ROM) and Secondary memory (Hard drive, CD, etc).

Primary Memory (RAM and ROM)



Random Access Memory (RAM)

- It is also called read-write memory or the main memory or the primary memory.

- The programs / instructions and data that the CPU requires during the execution of a program are stored in this memory.
- It is a volatile memory as the data is lost when the power is turned off.
- RAM is further classified into two types- SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory).

DRAM	SRAM
1. Constructed of tiny capacitors that leak electricity.	1.Constructed of circuits similar to D flip-flops.
2.Requires a recharge every few milliseconds to maintain its data.	2.Holds its contents as long as power is available.
3.Inexpensive.	3.Expensive.
4. Slower than SRAM.	4. Faster than DRAM.
5. Can store many bits per chip.	5. Can not store many bits per chip.

Read-Only Memory (ROM)

- Stores crucial information essential to operate the system, like the program essential to boot the computer.
- It is not volatile.
- Always retains its data.
- Used in embedded systems or where the programming needs no change.
- Used in calculators and peripheral devices.
- ROM is further classified into four types- PROM, EPROM, EEPROM.

Types of Read-Only Memory (ROM)

- **PROM (Programmable read-only memory)** – It can be programmed by the user. Once programmed, the data and instructions in it cannot be changed.
- **EPROM (Erasable Programmable read-only memory)** – It can be reprogrammed. To erase data from it, expose it to ultraviolet light. To reprogram it, erase all the previous data.
- **EEPROM (Electrically erasable programmable read-only memory)** – The data can be erased by applying an electric field, with no need for ultraviolet light. We can erase only portions of the chip.

Difference Between RAM & ROM

RAM	ROM
1. Temporary Storage.	1. Permanent storage.
2. Store data in MBs.	2. Store data in GBs.
3. Volatile.	3. Non-volatile.
4.Used in normal operations.	4. Used for startup process of computer.
5. Writing data is faster.	5. Writing data is slower.

Secondary Memory

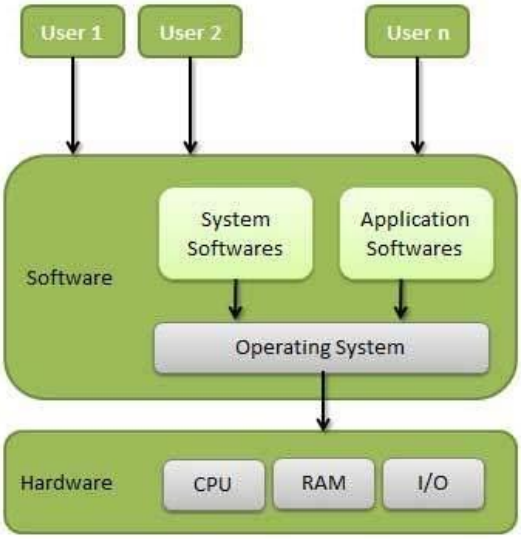
Secondary memory is computer memory that is non-volatile and persistent in nature and is not directly accessed by a computer/processor. Data in secondary memory must be copied into primary storage before use. Secondary memories are the slower and cheaper form of memory as compared to primary memory.

Secondary memory devices include:

- Magnetic disks like hard drives and floppy disks
- Magnetic tapes
- Optical disks such as CDROMs, DVDs, etc.

Operating System (OS)

Operating System (OS) provides the environment in which users can run their own program. It is system software that manages computer hardware, software resources, and provides common services for computer programs. An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- I/O Management

Types of Operating System

1. Multiprogramming Operating System

A multiprogramming operating system may run many programs on a single processor computer. If one program must wait for an input/output transfer in a multiprogramming operating system, the other programs are ready to use the CPU. As a result, various jobs may share CPU time.

Example: Windows O/S, UNIX O/S,

2. Multiprocessing Operating System

Multiprocessor Operating System refers to the use of two or more central processing units (CPU) within a single computer system.

Example: UNIX

3. Batch Operating System

Batch processing is a technique in which an Operating System collects the programs and data together in a batch before processing starts.

Examples: Payroll System, Bank Statements

4. Time-Sharing Operating Systems

Each task is given some time to execute so that all the tasks work smoothly. Each user gets the time of CPU as they use a single system. These systems are also known as Multitasking Systems.

Examples of Time-Sharing OSs are: Multics, Unix

5. Distributed Operating System

A distributed operating system uses many central processors to serve multiple real-time applications and users. As a result, data processing jobs are distributed between the processors.

Examples of Distributed Operating System are- LOCUS, Solaris

6. Network Operating System

These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions.

Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008

Processor

A Processor or Microprocessor is a small chip that resides in computers and other electronic devices. Its basic job is to receive input and provide the appropriate output. A processor is an integrated electronic circuit that performs the calculations that run a computer. A processor performs arithmetical, logical, input/output (I/O) and other basic instructions that are passed from an operating system (OS).

Processor Operations

The four primary functions of a processor are **fetch, decode, execute and write back.**

1. ### Fetch
   Every instruction has its own address and is stored in the main memory. The CPU fetches the address of the instruction which is to be executed from the program counter in the memory and performs the instruction.

2. ### Decode
   The instruction that is to be executed is converted into binary code so that the computer

can easily understand it and perform the required function. The process of conversion is known as decoding.

3. **Execute**

The process of performing the required task specified in the instruction is known as execution The execution of the instruction takes place in the CPU.

4. **Write back**

After performing the instruction the CPU store the result in the memory that process is known as a store or Write back.

I/O Devices

An input/output (I/O) device is any hardware used by a human operator or other systems to communicate with a computer. As the name suggests, input/output devices are capable of sending data (input) to a computer and receiving data from a computer (output).

Input devices:

An input device is any hardware that takes the input (usually from user), converts it into machine understandable binary language and sends it to the processor. Following are some of the important input devices which are used in a computer −

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Track Ball
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader(OMR)

Output device:

An output device is any peripheral that receives the (binary output) data from a processor (CPU), converts it to human understandable form and sends it usually for display, projection, or physical reproduction.

Example of Output Devices:

- Printer
- Headphones

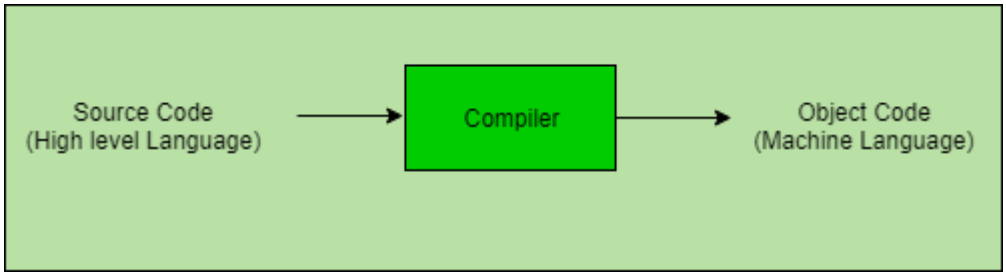
- Monitor
- Printer
- Speaker

Compiler

It is program that converts the code written in high level languages (like C, C++, etc.) into low level language (like assembly level language or machine level language).
Compiler is a language translator that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language.

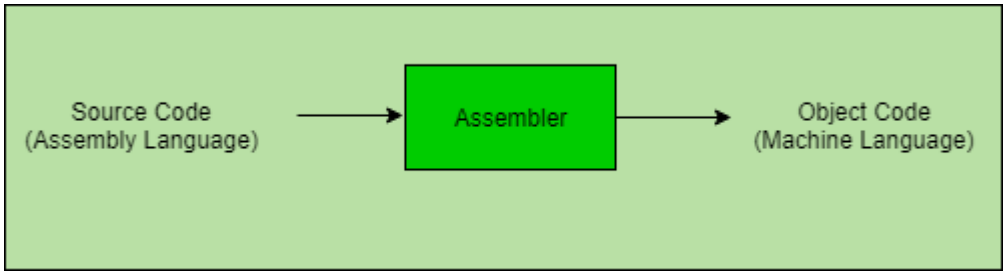
Example: C, C++, C#, Java.

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.



Assembler

The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.



Example of an Assembler: GAS, GNU, Turbo Assembler (TASM), GAS (GNU Assembler), MASM (Microsoft Assembler).

Assembly language

Assembly language is an extremely basic form of programming. It is a low level programming language for a computer. Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities. A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too complex for use by humans. So, the low-level assembly language is designed that represents various instructions in symbolic codes which are more understandable by humans.

Following are some examples of typical assembly language statements –

INC COUNT: Increment the memory variable

COUNT MOV TOTAL 48 : Transfer the value 48 in the memory variable TOTAL

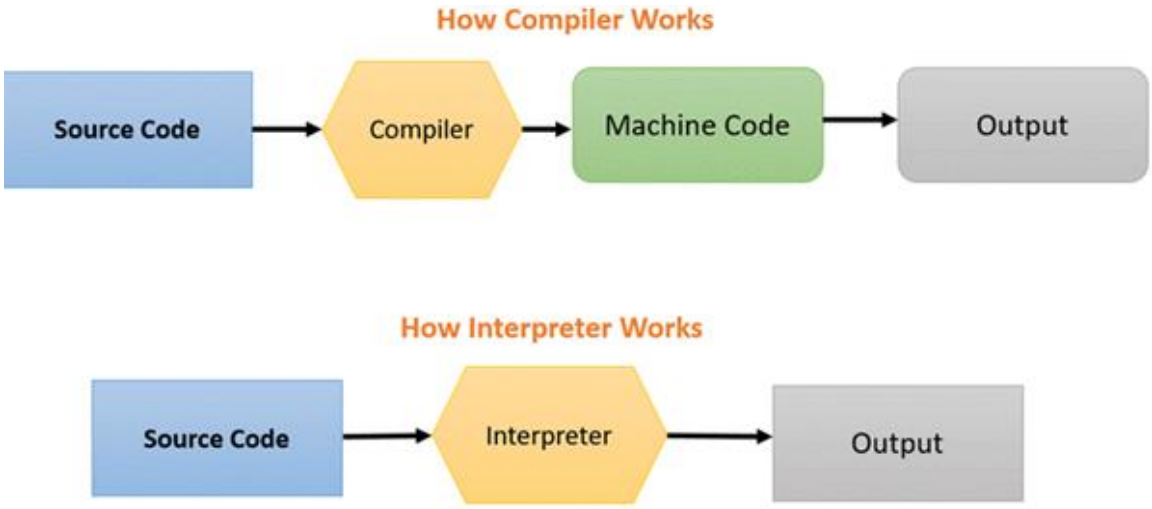
ADD AH, BH : Add the content of the BH register into the AH register

Interpreter

An interpreter is a computer program which translates and executes statements (written in high level language) line by line. It continues translating the program until the first error is met. As it gets first error, it stops. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.

For example, languages such as Python, LISP, Ruby, etc. use interpreters.

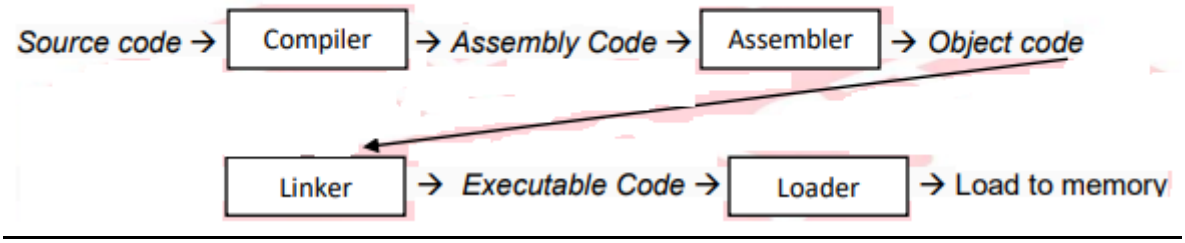
Difference Between Compiler & Interpreter



No	Compiler	Interpreter
1	Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input .
2	Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
3	Conditional Control Statements are Executes **faster**	Conditional Control Statements are Executes **slower**
4	**Memory Requirement : More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
5	Program need not be **compiled** every time	Every time higher level program is converted into lower level program
6	**Errors** are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
7	**Example** : C Compiler	**Example** : BASIC

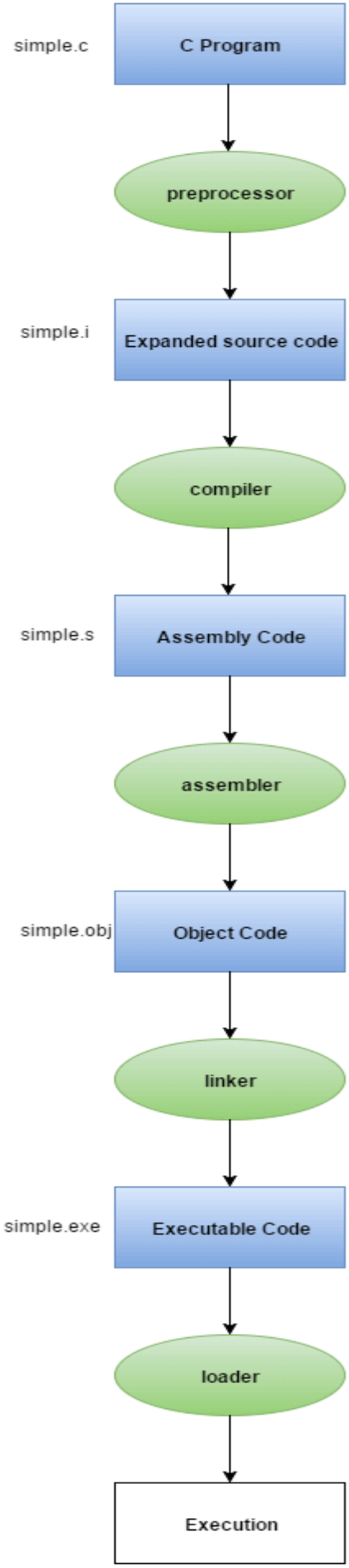
Linker and Loader

Linker and Loader are the utility programs that play a major role in the execution of a program. The Source code of a program passes through compiler, assembler, linker, loader in the respective order, before execution.

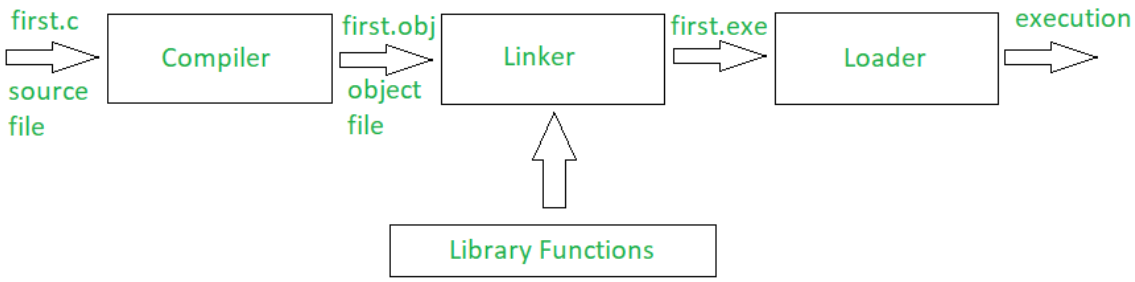


C Program Execution Process

1. C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.
2. Expanded source code is sent to compiler which compiles the code and converts it into assembly code.
3. The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.
4. The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code. A simple.exe file is generated.
5. The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console.


simple.c
C Program
preprocessor
simple.i
Expanded source code
compiler
simple.s
Assembly Code
assembler
simple.obj
Object Code
linker
simple.exe
Executable Code
loader
Execution

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



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.

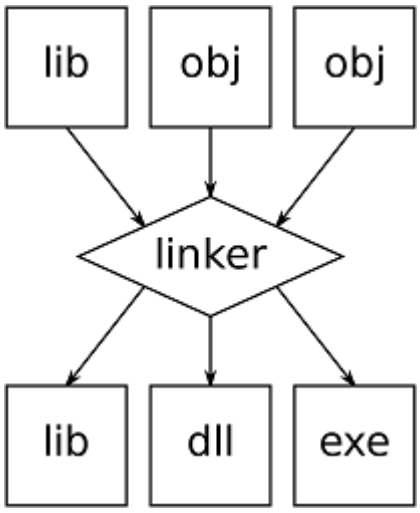
Linker:

Library functions are not a part of any C program but they are part of 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.

Linker is a program in a system which helps to link a object modules of program into a single object file. It performs the process of linking.

Linking is process of collecting and maintaining piece of code and data into a single file. Linker also link a particular module into system library. It takes one or multiple object files **from assembler & compiler as input and forms an executable file as output for loader.**

Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader. Linking is performed at the last step in compiling a program.

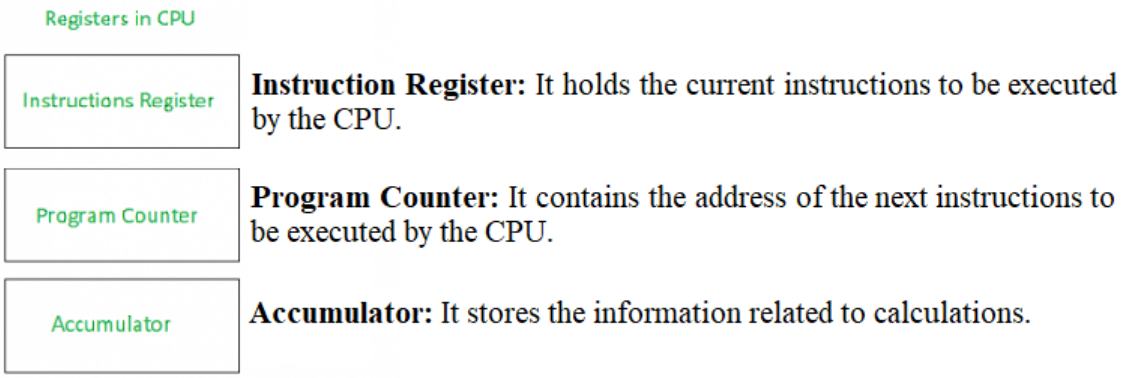


The source code is converted into machine code, and the linking is performed at the last step while compiling the program.

Source code -> compiler -> Assembler -> Object code -> Linker -> Executable file -> Loader

Loader:

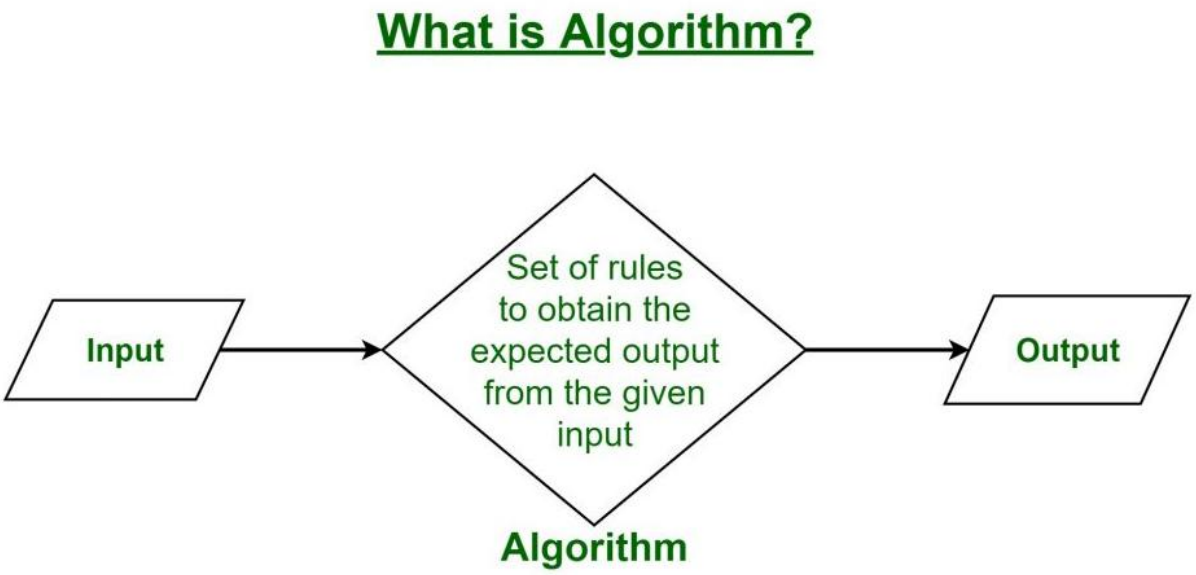
As the program that has to be executed must reside in the main memory of the computer, it is the responsibility of the loader to load the executable file of a program to the main memory for execution. It allocates the memory space to the executable module in main memory. The loader will load the .exe file in RAM and inform the CPU with the starting point of the address where this program is loaded.



The loader informs Program Counter about the first instruction and initiates the execution. Then onwards, Program Counter handles the task.

Algorithm

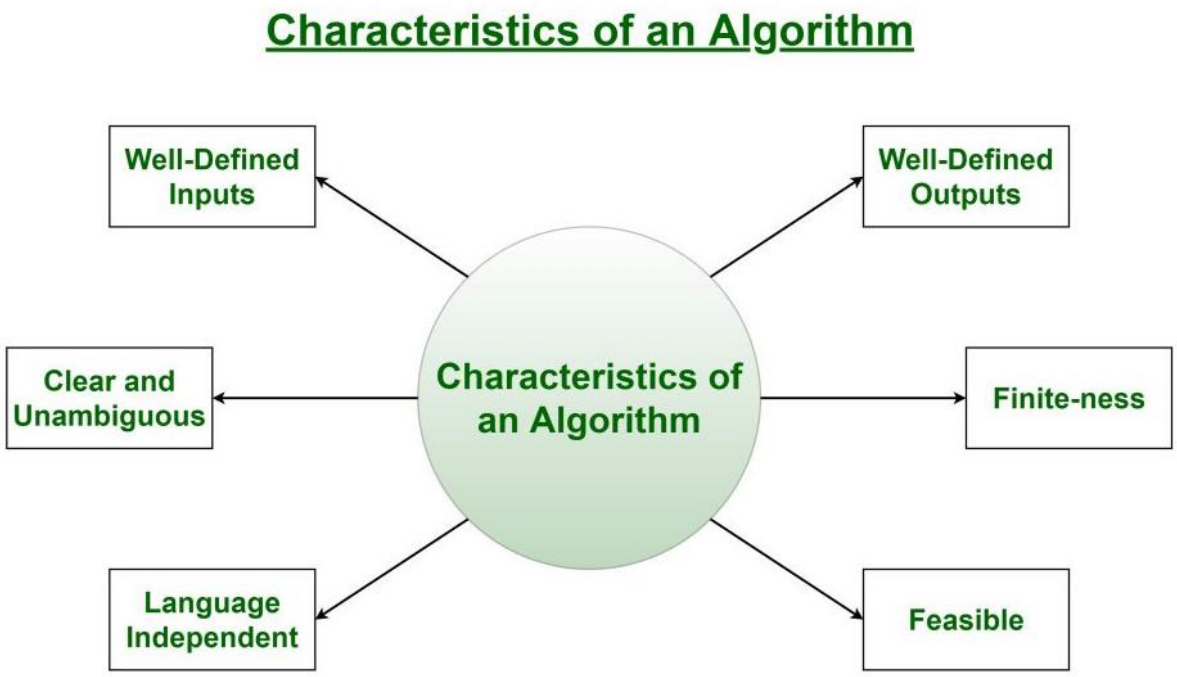
- The word **Algorithm** means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed in certain order to get the expected results.
- **Algorithm** is a step – by – step procedure which is helpful in solving a problem. If it is written in English like sentences then, it is called as 'PSEUDO CODE'.
- **Algorithm** is a step-by-step procedure which defines a set of instructions to be executed in a certain order to get the desired output.
- **Algorithm** is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output.



Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Characteristics of an Algorithm



- **Clear and Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finiteness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical such that it can be executed upon will the available resources.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

EXAMPLES

Algorithm to Add two numbers entered by the user

Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
sum←num1+num2
Step 5: Display sum
Step 6: Stop

Algorithm to calculate Gross Salary

Step 1:Start

Step 2:Read Basic Salary as basic

Step 3:Calculate Dearness Allowance as da=(10*basic)/100

Step 4:Calculate HouseRent Allowance as HRA=(12*basic)/100

Step 5:Calculate Gross Salary as gross_salary=basic+da+hra

Step 6:Print Gross Salary

Step 7:Stop

Algorithm to calculate Simple Interest

Step 1:Start

Step 2:Read Principal Amount, Rate and Time

Step 3:Calculate Interest using formula SI= ((principal_amount*rate*time)/100)

Step 4:Print Simple Interest

Step 5:Stop

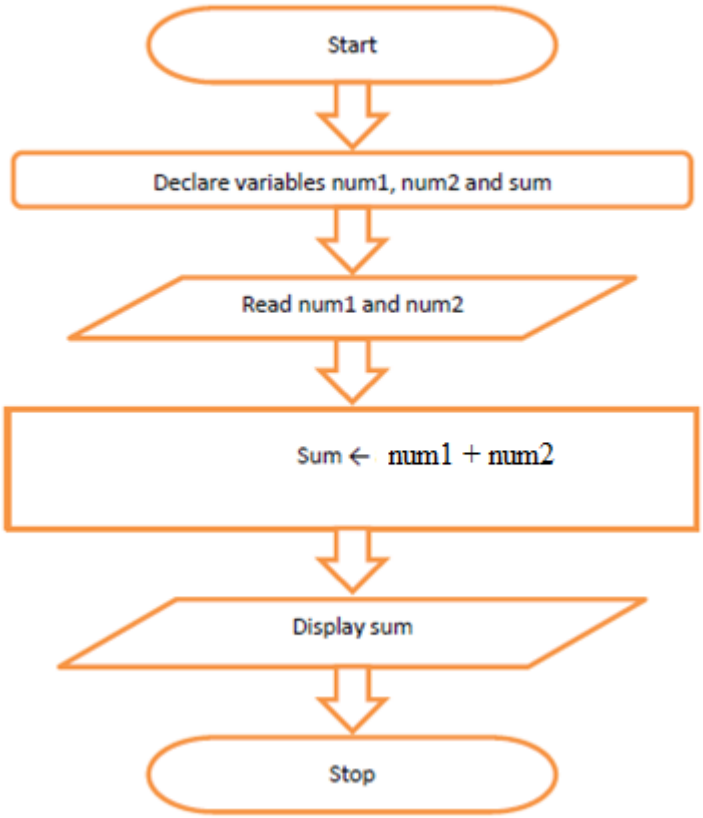
Flowchart

- Flowchart is a graphical representation of an algorithm.
- It is graphical or pictorial representation of workflow or process.
- It is step by step approach to solve a task or process in sequential order.
- Itis a diagrammatic representation of sequence of logical steps of a program.
- It uses various symbols to show the operations and decisions to be followed in a program.

Flowchart Symbols

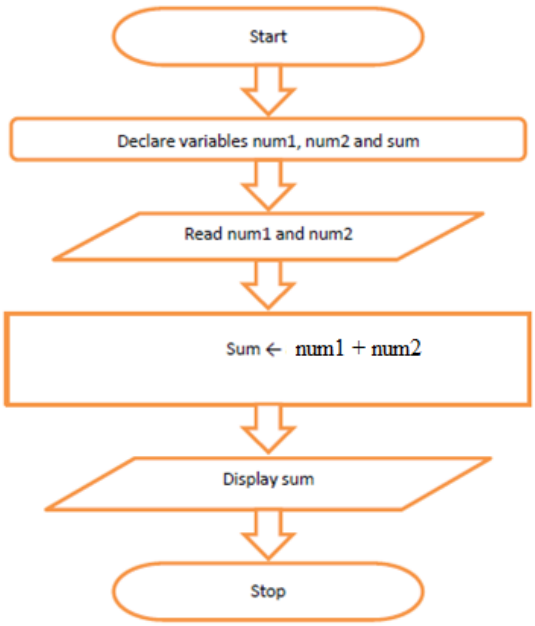
Symbol	Symbol Name	Purpose
	Start/Stop	Used at the beginning and end of the algorithm to show start and end of the program.
	Process	Indicates processes like mathematical operations.
	Input/ Output	Used for denoting program inputs and outputs.
	Decision	Stands for decision statements in a program, where answer is usually Yes or No.
	Arrow	Shows relationships between different shapes.
	On-page Connector	Connects two or more parts of a flowchart, which are on the same page.
	Off-page Connector	Connects two parts of a flowchart which are spread over different pages.

Flowchart to find the sum of two numbers entered by the user



Algorithm Vs Flowchart

Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
sum←num1+num2
Step 5: Display sum
Step 6: Stop



C Programming Basics

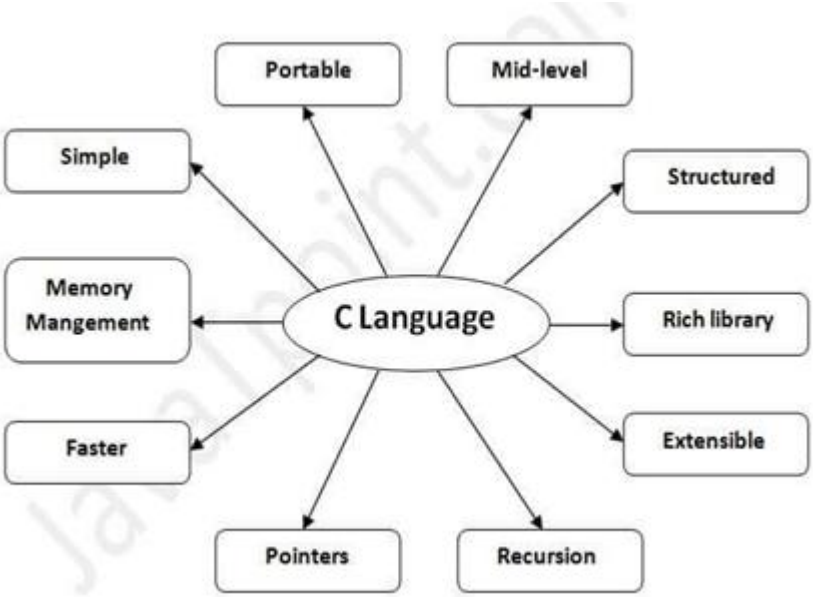
C is a high-level, general-purpose & procedural programming language developed at **AT& T's Bell Laboratories** of USA. It is a general-purpose, procedural, imperative computer programming language developed in **1972 by Dennis M. Ritchie** to develop the **UNIX operating system.**

C language combines the power of a **low-level language** and a **high-level language**. The low-level languages are used for system programming, while the high-level languages are used for application programming. It is because such languages are flexible and easy to use. Hence, C language is a widely used computer language.

Some Facts about C Language

- In **1988**, the **American National Standards Institute (ANSI)** had formalized the C language.
- C was invented to write **UNIX** operating system.
- C is a successor of 'Basic Combined Programming Language' (BCPL) called **B language.**
- **Linux OS**, **PHP**, and **MySQL** are written in C.

Features of C Language

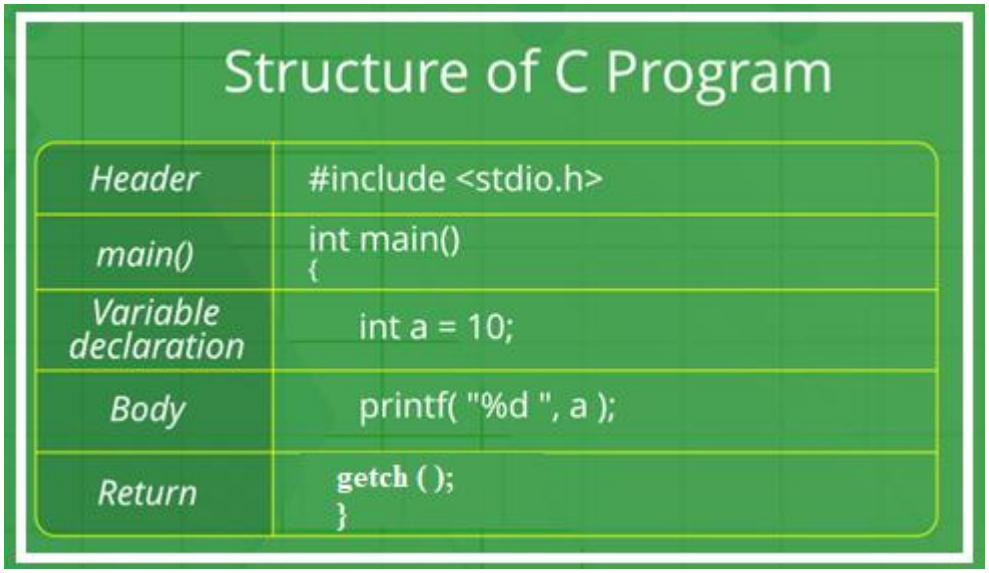


C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

Structure of a C program

The structure of a C program is as follows:



1. Header Files Inclusion

The first component is the inclusion of the Header files in a C program. A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.

Some of C Header files:

- stdio.h – Defines core input and output functions
- math.h – Defines common mathematical functions
- string.h – Defines string handling functions

2. main ( ) Method Declaration:

The next part of a C program is to declare the main( ) function.

Syntax to declare the main( ) method:

```
main( )
{
    }
```

3. Variable Declaration:

The next part of any C program is the variable declaration. No variable can be used without being declared. In a C program, the variables are to be declared before any operation in the function.

Example:

```
int a;
char abc;
```

4. Program Body

Here the set of instructions / lines of code is written according to the need of program according to the principles & syntax.

Example:

```
int main( )
{
   int a;

   printf("%d", a);

   getch ();
}
```

Syntax and logical errors in compilation

Error is an illegal operation performed by the user which results in abnormal working of the program.
Programming errors often remain undetected until the program is compiled or executed. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

There are mainly five types of errors exist in C programming:

- **Syntax error**
- **Run-time error**
- **Linker error**
- **Logical error**
- **Semantic error**

Syntax error

Syntax errors are also known as the compilation errors as they occurred at the compilation time. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language.

For example:

If we want to declare the variable of type integer,

1. **int** a; // this is the correct form
2. Int a; // this is an incorrect form.

Commonly occurred syntax errors are:

- If we miss the parenthesis ({}) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon (;) at the end of the statement.

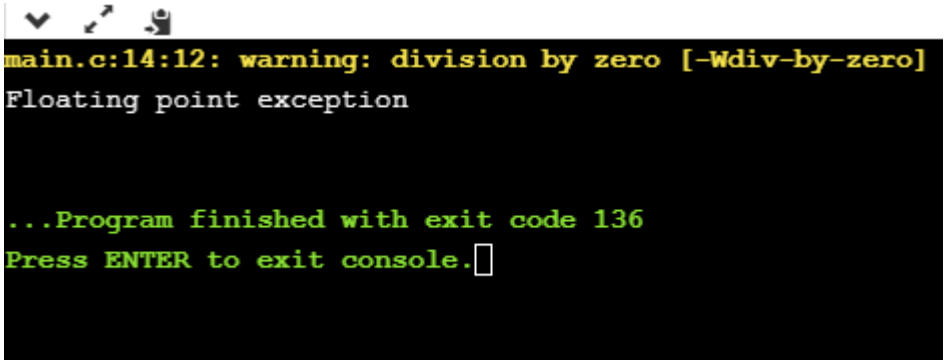
Run-time error

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

Example.

```
#include <stdio.h>
#include<conio.h>
int main()
{
 int a=2;
int b=2/0;
 printf("The value of b is : %d", b);
getch();
}
```

Output



Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file.

The most common linker error that occurs is that we use **Main()** instead of **main().**

Example.

```
#include <stdio.h>
#include<conio.h>
int Main()
{
 int a=78;
 printf("The value of a is : %d", a);
 getch();
}
```

Output

```
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Logical error

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors.

Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

- **Use of a un-initialized variable**
  int i;
  i=i+2;
- **Type compatibility**
  int b = "praveen";
- **Errors in expressions**
  int a, b, c;
  a+b = c;

Source Code , Object Code & Executable Code

Source Code:

The source code consists of the programming statements that are written by a programmer with the help text editor or a visual programming tool and then saved in a file.

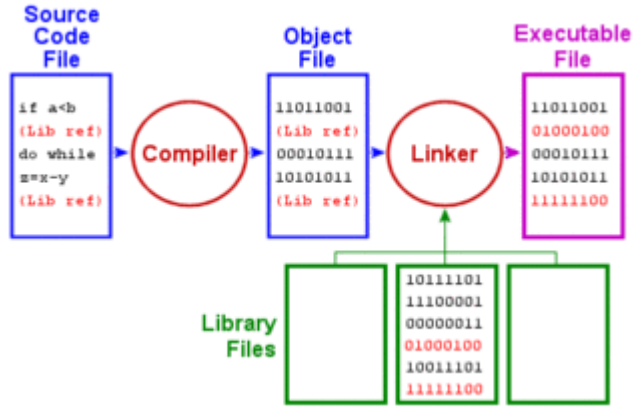
For example, a programmer using the C language, types in a desired sequence of C language statements using a text editor and then saves them with a file named "**praveen.c".** This file is said to contain the source code. It is now ready to be compiled with a C compiler.

Object Code:

When the source code file is compiled, it is translated into binary machine language code file which is called as Object Code File/Object File/Object Module. The object code file contains a sequence of instructions that the processor can understand . These object files may need to be linked against other object files, third party libraries.

Executable Code:

The object files/modules are not yet ready for execution. The object file is linked with other object files, third party libraries and runtime libraries. This linking is done by the Linker. After linking, the output of the linker is an executable code which is ready for execution by the processor and may be loaded into memory by the Loader.



The C Character Set

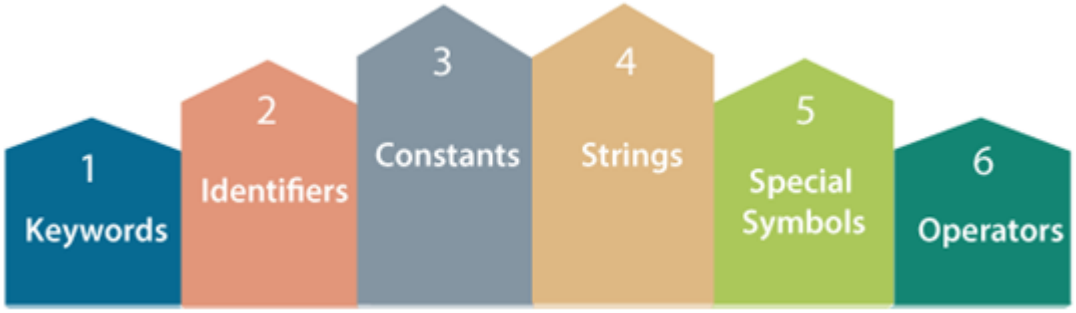
A character denotes any alphabet, digit or special symbol used to represent information.

Alphabets	A, B, ..., Y, Z a, b, ..., y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? / $

Tokens in C

- **Token** is defined as the smallest individual element in C.
- A token is the smallest element of a program that is meaningful to the compiler.
- We cannot create a program in C without using tokens in C.
- Tokens in C is the basic component for creating a program in C language.

Classification of tokens in C : Tokens in C language can be divided into the following categories:



- Keywords in C
- Identifiers in C
- Constant in C
- Strings in C
- Special Characters in C
- Operators in C

1. Keywords

Keywords are the words whose meaning has already been explained to the C compiler. Keywords are **predefined & reserved words** used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. The keywords cannot be used as variable names because if we do so, we are trying to assign a new meaning to the keyword, which is not allowed.

There are total 32 Keywords in C Programming language.

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

2. Identifiers in C

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

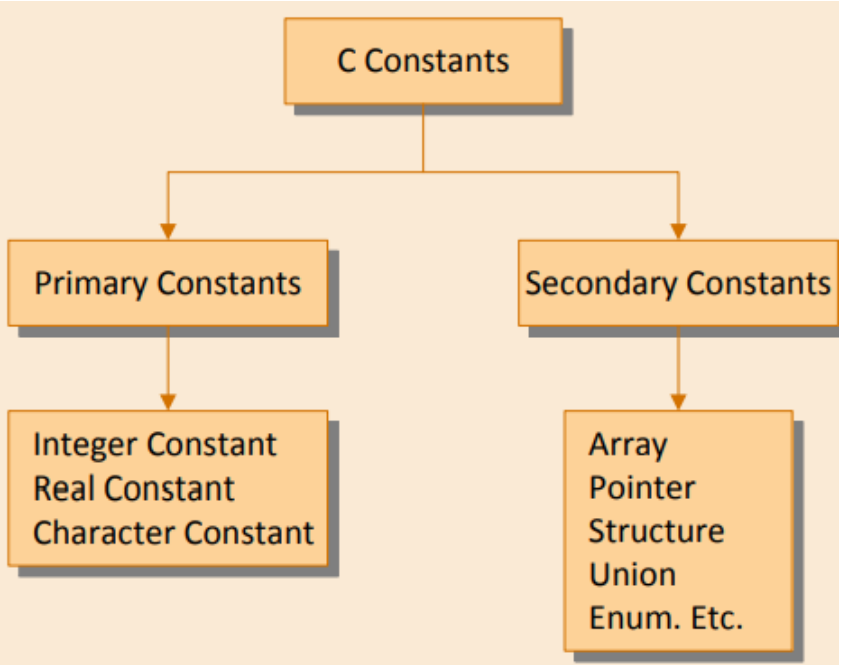
- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

3. Constants

A constant is an entity that doesn't change throughout the program.

Types of C Constants

1. Primary Constants
2. Secondary Constants



Rules for Constructing Integer Constants

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It can be either positive or negative. If there is no sign, then by default sign is positive.
4. No commas or blanks are allowed within an integer constant.
5. The allowable range for integer constants is **-32768 to +32767 (**For Turbo C or Turbo C++) **or -2147483648 to +2147483647(**for Visual Studio of GCC Compiler).

Example : 426 , +782, -8000 , -7605

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in **two forms—Fractional form and Exponential form**.

A. Fractional form

Rules for constructing real constants expressed in fractional form:

- **i.** A real constant must have at least one digit.
- **ii.** It must have a decimal point.
- **iii.** It could be either positive or negative.
- **iv.** Default sign is positive.
- **v.** No commas or blanks are allowed within a real constant.

Ex.: +325.34
426.0
-32.76
-48.5792

B. Exponential form.

The exponential form is usually used if the value of the constant is either too small or too large. In exponential form the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent.

Example : 0.000342 can be written in exponential form as **3.42e-4** (which in normal arithmetic means 3.42 x 10-4).

Rules for constructing real constants expressed in exponential form:

- **i.** The mantissa part and the exponential part should be separated by a letter e or E.
- **ii.** The mantissa part may have a positive or negative sign.
- **iii.** Default sign of mantissa part is positive.
- **iv.** The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
- **v.** Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.

Ex.: +3.2e-5
4.1e8
-0.2E+3
-3.2e-5

Rules for Constructing Character Constants

- **i.** A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
- **ii.** Both the inverted commas should **point to the left**. For example, ’A’ is a valid character constant whereas ‘A’ is not.

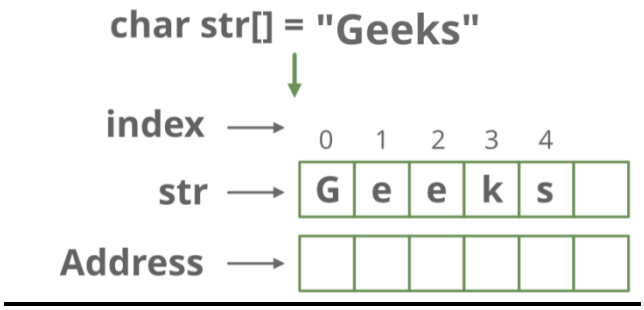
Ex.: ’A’
’I’
’5’
’=’

4.Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

charstr[6] = "Geeks"; // The compiler allocates the 6 bytes to the 'str' array.



5. Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf( ) is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

6. Operators in C

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

- **Unary Operator: The** Unary operator is an operator applied on single operand.
- **Binary Operator: The** binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators
- Assignment Operator

Variables

- A variable is a name given to a storage area or memory location that our programs can manipulate.
- A variable is a name of the memory location that is used to store the data. Its value can be changed, and it can be reused many times.
- Each variable in C has a specific type, which determines the size and layout of the variable's memory.

Rules for Variable Declaration

i. A variable name is any combination of 1 to 31 alphabets, digits or underscores. Some compilers allow variable names whose length could be up to 247 characters.
ii. The first character in the variable name must be an alphabet or underscore (_).
iii. No commas or blanks are allowed within a variable name.
iv. No special symbol other than an underscore can be used in a variable name.

Ex.: si_int
m_hra
pop_e_89

Syntax to declare a variable:

type variable_list ;

Here, **type** must be a valid C data type including char, int, float, double, bool and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

Example :

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

```
int a;
float b;
char c;
int _ab;
int a30;
```

There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code.

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

main(){

/* local variable declaration */
int a, b;
int c;

/* actual initialization */
  a =10;
  b =20;
  c = a + b;

printf("value of a = %d, b = %d and c = %d\n", a, b, c);

getch ( );
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

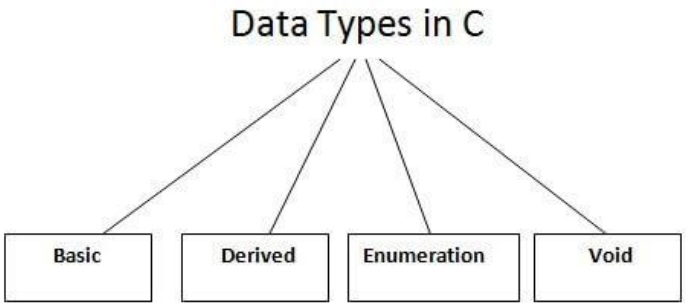
A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program

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

```
#include <conio.h>

/* global variable declaration */
int g;

main()
{

/* local variable declaration */
int a, b;

/* actual initialization */
  a =10;
  b =20;
  g = a + b;

printf("value of a = %d, b = %d and g = %d\n", a, b, g);

getch ( );
}
```

Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc. Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it.



There are the following data types in C language.

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf
Note: The sizes and ranges of int, short and long are compiler dependent. Sizes in this figure are for 16-bit compiler.			

Storage Class

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable.

Storage class of a variable tells us:

(a) Default Initial Value: What will be the initial / default initial value of the variable
(b) Storage: Where the variable would be stored.
(c) Scope: What is the scope of the variable; i.e. in which functions the value of the variable would be available.
(d) Lifetime: What is the life of the variable; i.e. how long would the variable exist.

Scope of Variable:

i. A scope is a region of the program & refers to **the area of the program where the variable is available & it can be accessed after its declaration.**
ii. Scope of a variable is the block of code in the entire program where the variable is declared, used, and can be modified.
iii. Scope is a section or region of a program where a variable has its existence; moreover
iv. Variable cannot be used or accessed beyond its Scope.

Lifetime of Variable:

i. The lifetime of a variable is **the time during which the variable stays in memory and is therefore accessible during program execution**.
ii. It is **the period of time during which it is available for use**.
iii. It represents the period of time during which it can hold a value.
iv. It defines the duration for which the computer allocates memory for it

There are four storage classes in C:

(a) Automatic storage class
(b) Register storage class
(c) Static storage class
(d) External storage class

Automatic Storage Class

For Automatic storage class auto keyword is used as: **auto int i ;**
The features of variable defined as an automatic storage class are:

Storage:	Memory.
Default value:	An unpredictable value, often called a garbage value.
Scope:	Local to the block in which the variable is defined.
Life:	Till the control remains within the block in which the variable is defined.

Default Initial value of an automatic variable is illustrated in the following program.

```
# include <stdio.h>
#include <conio.h>
main ( )
{
auto int i, j ;
printf ( "%d %d\n", i, j ) ;
getch ( );
}
```

The output of the above program could be...
1211 221
Where, 1211 and 221 are garbage values of i and j.

Scope and life of an automatic variable is illustrated in the followingprogram.

```
# include <stdio.h>
#include <conio.h>
main( )
{
auto int i = 1 ;
{
auto int i = 2 ;
{
auto int i = 3 ;
printf ( "%d ", i ) ;
}
printf ( "%d ", i ) ;
}
printf ( "%d\n", i ) ;
getch( ) ;
}
```

The output of the above program would be:
3 2 1

Register Storage Class

For Register Storage class, register keyword is used as : register int sum ;

The features of variable defined as register storage class are as

Storage:	CPU registers.
Default Initial value:	Garbage value.
Scope:	Local to the block in which the variable is defined.
Lifetime:	Till the control remains within the block in which the variable is defined.

Static Storage Class

For Static storage class, static keyword is used as : **static int i = 1 ;**
The features of a variable defined to have a **static** storage class are asunder:

Storage:	Memory.
Default value:	Zero.
Scope:	Local to the block in which the variable is defined.
Lifetime:	Value of the variable persists between different function calls.

```
#include <stdio.h>
void increment( ) ;
int main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
    return 0 ;
}
void increment( )
{
    auto int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

```
#include <stdio.h>
void increment( ) ;
int main( )
{
    increment( ) ;
    increment( ) ;
    increment( ) ;
    return 0 ;
}
void increment( )
{
    static int i = 1 ;
    printf ( "%d\n", i ) ;
    i = i + 1 ;
}
```

The output of the above programs would be:

```
1
1
1
```

```
1
2
3
```

External Storage Class

For External Storage class, variable is declared as : **extern int y ;**
The features of a variable whose storage class has been defined as external are as follows:

Storage: Memory.
Default value: Zero.
Scope: Global.
Lifetime : As long as the program's execution doesn't come to an end.

```
include<stdio.h>
include<conio.h>
int x = 21 ;
main( )
{
extern int y ;
printf ( "%d %d\n", x, y ) ;
getch ( );
}
int y = 31 ;
```

Here, x and y both are global variables. Since both of them have been defined outside all the functions both are external storage class.