UNIT I

UNIT I Introduction to Programming and Problem Solving
History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program- Algorithms, flowcharts (Using Dia Tool), pseudo code. Introduction to C-Programming Data Types, Variables, and Constants, Operators & Expressions ,Basic Input and Output, Operations, Type Conversion, and Casting.

Introduction to Computers

Computer:

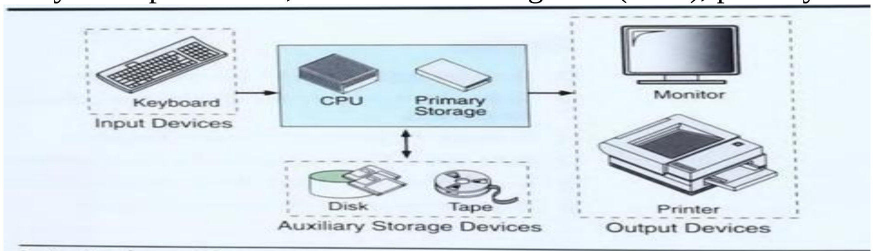
- Computer is an electronic device which takes input data, process that data and gives desired output to the user.
- **Charles Babbage** is called as **father of computer**.
- We use computers to get **accurate results** and **fast results**.

Computer Systems:

- A computer system is made up of two components. They are software and hardware.
- The computer **hardware** is physical components.
- The computer **software** is collection of programs(instructions) that allow the hardware to do its job.

Computer hardware:

- The hardware components of the computer system consist of 5 parts.
- They are inputdevices, Central Processing Unit (CPU), primary storage, auxiliary storage and output devices.



Basic Hardware Components

- The **input device** is usually a **<u>keyboard</u>** where programs and data are entered into the computer. Examples of other input devices are **mouse, a pen or stylus, a touchscreen or an audio input unit.**
- The **Central Processing Unit (CPU)** is responsible for executing instructions such as **arithmetic calculations, comparisons among data, movement of data inside the system**. Todays computers have one, two or more CPU's.
- **Primary Storage,** is also known as **main memory**, is a place where the programs and data are temporarily stored during processing. The data in the primary storage is ***erased*** when we turnoff a computer.
- **Auxiliary storage,** is also known as **secondary storage**, is used for both input and output. In this storage both **programs and data stored permanently**. Even when we turn off the computer also data remain in the secondary storage.
- **Output device** is usually a **<u>monitor</u>** or a **<u>printer</u>** to show output. If the output is shown on the monitor it is called **soft copy** and if it is printed on the printer, we say it is as a **hard copy**.

Computer software:

- Computer software is divided into two broad categories. They are **System Software** and **Application Software.**

System software:

- System software manages the computer resources.
- Operating System is a system software.
- It provides the **interface** between hardware & the user and but **does nothing to directly serve the user's needs**.

- It provides a platform for the application software to run.

Application Software:

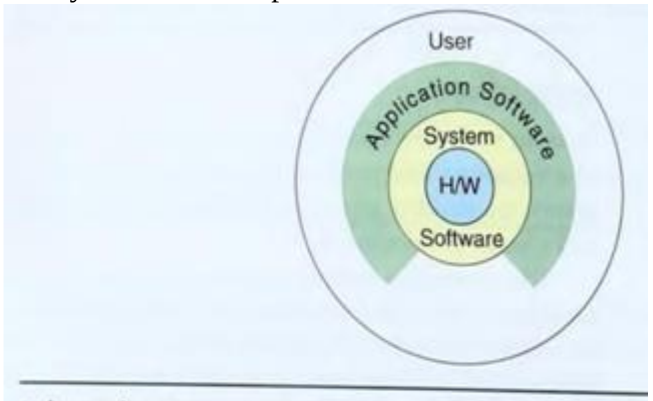
It is a collection of programs written for a specific application.

Eg: M.S.Office, Windows media players, Web browsers etc.

- Application software on the other hand is **directlyresponsibleforhelpingusers** to solve their problems.

The relationship between system software and application software can be shown below.

- In the diagram each circle represents an interface point.
- To work with a system, user uses an application software.
- The application software interacts with the operating system which is part of the system software layer.
- The system software provides the direct interaction with the hardware.



Relationship Between System and Application Software

History of Computers:

Generations of Computers:

A generation of computers refers to the specific improvements in computer technology with time. In 1946, electronic pathways called circuits were developed to perform the counting. It replaced the gears and other mechanical parts used for counting in previous computing machines.

In each new generation, the circuits became smaller and more advanced than the previous generation circuits. The miniaturization helped increase the speed, memory and power of computers. There are five generations of computers which are described below;

First Generation Computers:

The first generation (1946-1959) computers were slow, huge and expensive. In these computers, vacuum tubes were used as the basic components of CPU and memory. These computers were mainly depended on batch operating system and punch cards. Magnetic tape and paper tape were used as output and input devices in this generation; Some of the popular first generation computers are

- **ENIAC** (Electronic Numerical Integrator and Computer)
- **EDVAC** (Electronic Discrete Variable Automatic Computer)
- **UNIVACI**(Universal Automatic Computer)
- **IBM-701**
- **IBM-650**

Second Generation Computers:

The second generation (1959-1965) was the era of the transistor computers. These computers used transistors which were cheap, compact and consuming less power; it made transistor computers faster than the first generation computers.

In this generation, magnetic cores were used as the primary memory and magnetic disc and tapes were used as the secondary storage. Assembly language and programming languages like COBOL and FORTRAN, and Batch processing and multiprogramming operating systems were used in these computers.

Some of the popular second generation computers are;

- **IBM 1620**
- **IBM 7094**
- **CDC 1604**
- **CDC 3600**
- **UNIVAC 1108**

Third Generation Computers:

The third generation computers used integrated circuits (ICs) instead of transistors. A single IC can pack huge number of transistors which increased the power of a computer and reduced the cost. The computers also became more reliable, efficient and smaller in size. These generation computers used remote processing, time-sharing, multi programming as operating system. Also, the high-level programming languages like FORTRON-II TO IV, COBOL, PASCAL PL/1, ALGOL-68 were used in this generation.

Some of the popular third generation computers are;

- **IBM-360 series**
- **Honeywell-6000 series**
- **PDP(Personal Data Processor)**
- **IBM-370/168**
- **TDC-316**

Fourth Generation Computers:

The fourth generation (1971-1980) computers used very large scale integrated (VLSI) circuits; a chip containing millions of transistors and other circuit elements. These chips made this generation computers more compact, powerful, fast and affordable. These generation computers used real time, time sharing and distributed operating system. The programming languages like C, C++, DBASE were also used in this generation.

Some of the popular fourth generation computers are;

- **DEC 10**
- **STAR 1000**
- **PDP 11**
- **CRAY-1(Super Computer)**
- **CRAY-X-MP(Super Computer)**

Fifth Generation Computers:

In fifth generation (1980-till date) computers, the VLSI technology was replaced with ULSI (Ultra Large Scale Integration). It made possible the production of microprocessor chips with ten million electronic components. This generation computers used parallel processing hardware and AI (Artificial Intelligence) software. The programming languages used in this generation were C, C++, Java, .Net, etc.

Some of the popular fifth generation computers are;

- **Desktop**
- **Laptop**
- **NoteBook**
- **UltraBook**
- **ChromeBook**

Basic organization of a computer: ALU, input-output units, memory, program counter:

A Central Processing Unit is the most important component of a computer system. A CPU is a hardware that performs data input/output, processing and storage functions for a computer system.

A computer consists of 5 functionally independent main parts:

1) input, 2)memory,3)arithmetic & logic unit(ALU) 4)output and 5)control units.

Input Unit :

The input unit consists of input devices such as a mouse, keyboard, scanner ,joystick etc.

These devices are used to input information or instruction into the computer system. That is that input unit takes data from us to the computer in an organized for processing.

The input unit performs the following major functions:

-The input unit converts the inputted data or instructions into binary form for further processing.

-Input unit transmits the data to the main memory of the computer.

Output Unit:

Output unit consists of devices that are used to display the results or output of processing. The output data is first stored in the memory and then displayed in human readable form through output devices.

Examples : monitor, printer, projector, graphic displays etc.

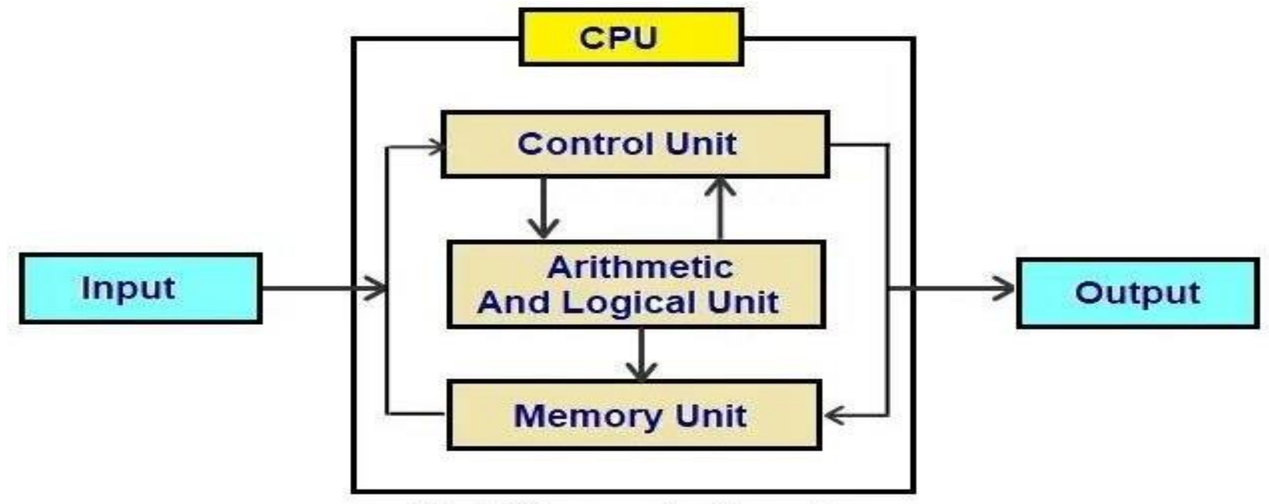
The output unit performs the following major functions:

-The output unit accepts the data or information in binary form from the main memory of the computer system.

-The output unit converts the binary data into a human –readable form for better understanding.

CPU(Central Processing Unit):

CPU is known as the brain of the computer system. It is an electronic hardware device that processes all the operations (eg: arithmetic and logical operations) of the computer. In other words, all the major calculations , operations or comparisons are performed inside the CPU.



Block Diagram of a Computer

Control Unit:
The control unit of a CPU controls all the activities and operations of the computer. It is also responsible for controlling input/output , memory and other devices connected to the CPU.
The control unit acts like the supervisor which determines the sequence in which computer programs and instructions are executed.

ALU (Arithmetic Logic Unit) :
ALU (Arithmetic Logic Unit) is responsible for performing arithmetic and logical functions or operations. It consists of two subsections, which are:

- Arithmetic Section
- Logic Section
- **Arithmetic Section:** By arithmetic operations, we mean operations like addition, subtraction, multiplication, and division, and all these operation and functions are performed by ALU. Also, all the complex operations are done by making repetitive use of the mentioned operations by ALU.
- **Logic Section:** By Logical operations(AND ,OR etc), we mean operations or functions like selecting, comparing, matching, and merging the data, and all these are performed by ALU.
- Note: CPU may contain more than one ALU and it can be used for maintaining timers that help run the computer system.

Memory Unit:
Memory Unit is an essential part of the computer system which is used to store data and instructions before and after processing.
There are two types of memory units:
1. Primary memory/Main memory
2. Secondary memory/Auxiliary memory

Primary Memory: Primary memory cannot store a vast amount of data.
The data stored in the primary memory is temporary. The data will be lost if they are disconnected from the power supply.
Primary-storage is a fast-memory that operates at electronic-speed. Programs must be stored in the memory while they are being executed.
Primary memory is also known as the Main Memory or temporary memory. Random Access Memory (RAM) is an example of Primary memory.

Secondary Memory:
Secondary memory is also known as the Auxiliary Memory or permanent storage.
Secondary-storage is used when large amounts of data & many programs have to be stored.
The data stored in the secondary memory is safe even when there is a power failure or no power supply. Hard Disk is usually considered a secondary memory.
Eg: Magnetic disks and optical disks(CD-ROMs).

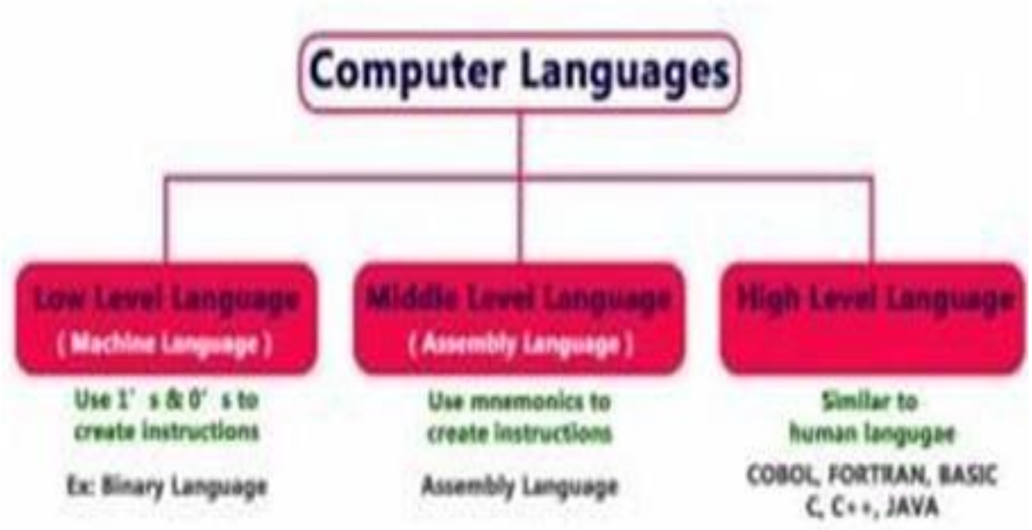
NOTE: Primary memory is the only memory that is directly accessible to the CPU. Secondary memory is not directly accessible to the CPU. The data accessed from the secondary unit is first loaded into RAM and then further transferred to the processing unit.

Program Counter:

A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory.

A program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 1. After each instruction is fetched, the program counter points to the next instruction in the sequence. When the computer restarts or is reset, the program counter normally reverts to 0.

Computer Languages Classification



- Computer languages have been evolved from Low Level to High Level Languages. In the earliest days of computers, only Binary Language was used to write programs.
- The computer languages are classified as follows...

Low Level Language (Machine Language)

- Low Level language is the only language which can be understood by the computer. . Low level language is also known as Machine Language. Binary Language is an example of low level language.
- The binary language contains only two symbols 1 & 0. All the instructions of binary language are written in the form of binary numbers 1's & 0's.
- As the CPU directly understands the binary language instructions, it does not require any translator.
- CPU directly starts executing the binary language instructions, and takes very less time to execute the instructions as it does not requires any translation. Low level language is considered as the First Generation Language (FGL)

Advantages

- A computer can easily understand the low level language.
- Low level language instructions are executed directly without any translation.
- Low level language instructions require very less time for their execution.

Disadvantages

- Low level language instructions are very difficult to use and understand.
- Low level language instructions are machine dependent, that means a program written for particular machine does not executes on other machine.
- In low level language, there is more chance for errors and it is very difficult to find errors, debug

and modify.

Middle Level Language (Assembly Language)

- Middle level language is a computer language in which the instructions are created using symbols such as letters, digits and special characters. Assembly language is an example of middle level language.
- In assembly language, we use predefined words called mnemonics. Binary code instructions in low level language are replaced with mnemonics and operands in middle level language.
- But computer cannot understand mnemonics, so we use a translator called Assembler to translate mnemonics into binary language.

Advantages

- Writing instructions in middle level language is easier than writing instructions in low level language.
- Middle level language is more readable compared to low level language.
- Easy to understand, find errors and modify.

Disadvantages

- Middle level language is specific to a particular machine architecture that means it is machine dependent.
- Middle level language needs to be translated into low level language.
- Middle level language executes slower compared to low level language.

High Level Language

- High level language is a computer language which can be understood by the users.
- High level language is very similar to the human languages and have a set of grammar rules that are used to make instructions more easily.
- Every high level language have a set of predefined words known as Keywords and a set of rules known as Syntax to create instructions.
- High level language needs to be converted into low level language to make it understandable by the computer. We use Compiler or interpretor to convert high level language to low level language.
- Languages like COBOL, FORTRAN, BASIC, C ,C++, JAVA etc., are the examples of high level languages.

Advantages

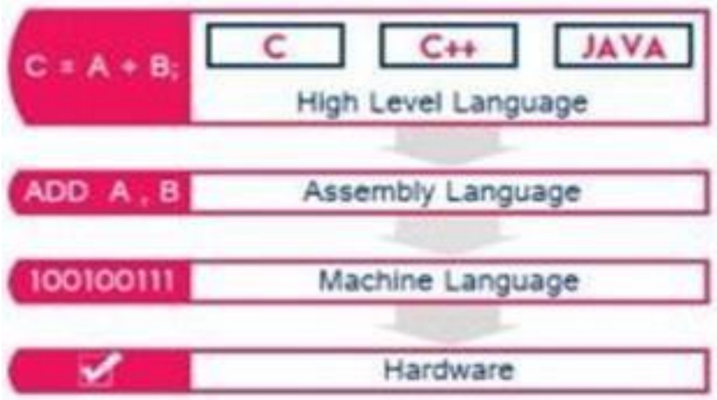
- Writing instructions in high level language is more easier.
- High level language is more readable and understandable.
- The programs created using high level language runs on different machines with little change or no change.
- Easy to understand, create programs, find errors and modify.

Disadvantages

- High level language needs to be translated to low level language.
- High level language executes slower compared to middle and low level languages

Compiler	**Interpreter**
Compiler Takes **Entire** program as input	Interpreter Takes **Single** instruction as input
Intermediate Object Code is **Generated**	**No** Intermediate Object Code is **Generated**
Conditional Control Statements executes **Faster**	Conditional Control Statements executes **Slower**
Memory Requirement: **More** (Since Object Code is Generated)	**Memory Requirement** is **Less**
Program need not be **compiled** every time	Every time higher level program is converted into lower level program
Errors are displayed after **entire program** is checked	**Errors** are displayed for **every instruction** interpreted (if any)
Example: C Compiler	**Example**: BASIC

Understanding Computer Languages
The following figure provides few key points related to the computer languages.



From the above figure, we can observe the following key points...

- The programming languages like C, C++, Java etc., are written in High level language which is more comfortable for the developers.
- High level language is more closer to the users.

Introduction to C Programming Language :

- C is a Structured Programming Language. C is derived from ALGOL, the first language to use a block structure.
- In 1967, Martin Richards developed a language called Basic Combined programming language (BCPL).
- Ken Thompson followed in 1970 with a similar language called B. B was used to develop the first version of UNIX.
- Dennis Ritchie developed C which took many concepts from ALGOL, BCPL.
- In 1972 first version is release. but got popular with the book by Brian W. Kernighan and Dennis Ritchie in 1978
- In 1983, the American National standards Institute (ANSI) started certifying the standard C. It was approved in the Dec-1989.
- In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C is knownas C89.
- In 1995, minor changes were made to the standard. This version is known as *C95*.
- A much more significant update was made in 1999. The changes incorporated into the standard, now known as *C99* are summarized in the following list.
    1. Extensions to the character type to support non-English characters.
    2. A Boolean type
    3. Extensions to the integer type
    4. Inclusion of type definitions in the for statement.
    5. Addition of imaginary and complex types.
    6. Incorporation off the C++ style line comment (///)

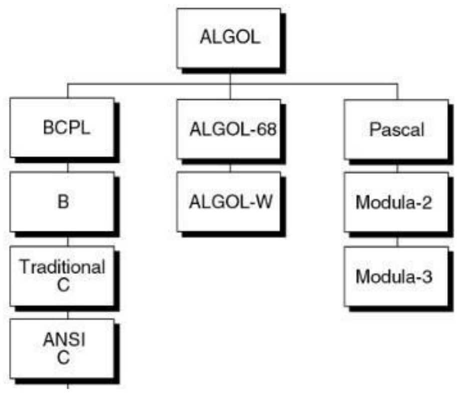


Figure: Taxonomy of C Language

Structure of C program:

The program written in C language follows this basic structure. The sequence of sections should be as they are in the basic structure. A C program should have one or more sections but the sequence of sections is to be followed.

1. Documentation section
2. Linking section
3. Definition section
4. Global declaration section
5. Main function section

   {

   Declaration section
   Executable section
   }
6. Sub program or function section

1. Documentation Section:-

This comes first and is used to document the use of logic or reasons in your program. It can be used to write the program's objective, developer and logic details. The documentation is done in C language with /* and */. Whatever is written between these two are calledcomments.

2. Linking Section:-

This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.

e.g. #include <stdio.h>

3. Definition Section:-

It is used to declare some constants and assign them some value.

e.g. #define MAX 25

Here #define is a compiler directive which tells the compiler whenever MAX is found in the program, replace it with 25.

4. **Global Declaration Section:-**

Here the variables which are used throughout the program (including main and other functions) are declared so as to make them global(i.e accessible to all parts of program)

e.g. int i; (before main())

5. **Main Function Section :-**

It tells the compiler where to start the execution from.

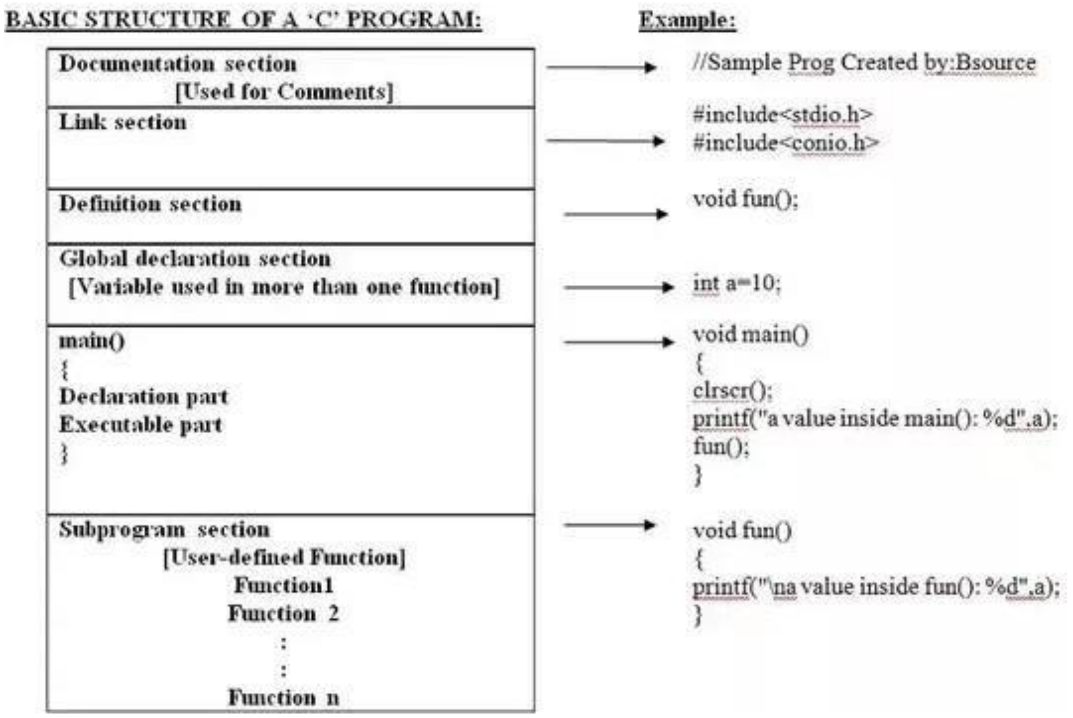
```
main()
{
point from where execution starts
}
```

main function has two sections

1. Declaration section: In this, the variables and their data types are declared.
2. Executable section: This has the part of program which actually performs the task we need.

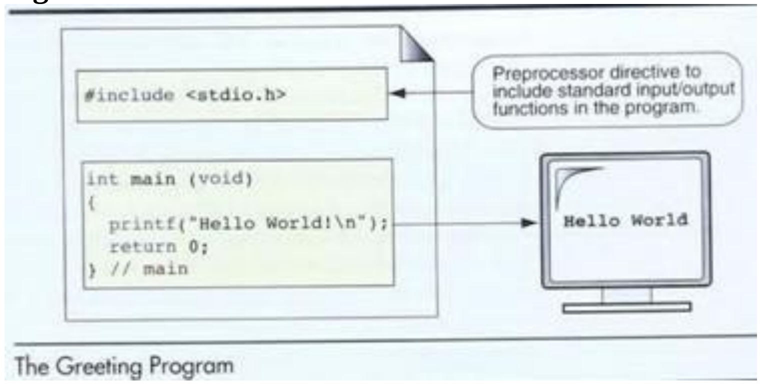
6. **Sub Program Or Function Section:-**

This has all the sub programs or the functions which our program needs.


BASIC STRUCTURE OF A 'C' PROGRAM:
Example:
Documentation section [Used for Comments]
Link section
Definition section
Global declaration section [Variable used in more than one function]
main() { Declaration part Executable part }
Subprogram section [User-defined Function] Function1 Function 2 : : Function n

```
//Sample Prog Created by:Bsource
#include<stdio.h>
#include<conio.h>
void fun();
int a=10;
void main()
{
clrscr();
printf("a value inside main(): %d",a);
fun();
}
void fun()
{
printf("\na value inside fun(): %d",a);
}
```

Your first C Program



```
#include <stdio.h>

int main (void)
{
  printf("Hello World!\n");
  return 0;
} // main
```

Preprocessor directive to include standard input/output functions in the program.
Hello World

The Greeting Program

- Our first C program is very simple. it has only one preprocessor command, no global declarations, and no local definitions. Its purpose will be simply to print a greeting to the user.
- So, the statements section had only 2 statements. one that prints a greeting and one that stops the program.

Preprocessor commands:

- Preprocessor commands come at the beginning of the program. All preprocessor commands start with a pound sign (#).
- The preprocessor command tells the compiler to include the standard input/output library file in the program.
- We need this library file to print a message to the terminal. Printing is one of the input/output processes identified in this library.
- The complete syntax for this command is shown below.

  #include <stdio.h>
- A preprocessor command it must start with a pound symbol, there can be no space between the pound sign and the keyword
- **include**. it tells the preprocessor that we want the library include, it tells the preprocessor that we want the library
- The name of the header file is **stdio.h,** the abbreviation is standard input/ output header file.

main():

- The executable part of our program begins with the function main, which is identified by the function header.
- In the main function **int** says that the function will return an integer value to the OS, that the functions name is main, and it has no parameters (the parameters list is void).

  int main(void)
- Within main there are 2 statements.
  - one to print the message and
  - one to terminate the program.
- The print statement uses a library function to execute this print function, **printf**, followed by a parameters list enclosed in parentheses.
- For example, whatever we want to display we place that in two double quote marks (".."). the \n at the end of the message tells the computer to advance to the next line in the output.
- The last statement in our program, return 0, terminates the program and returns control to the OS.

COMMENTS: C uses two different types of comments:

Comments are the non executable statements in a program.

The compiler ignores these comments when it translates the program into executable code.

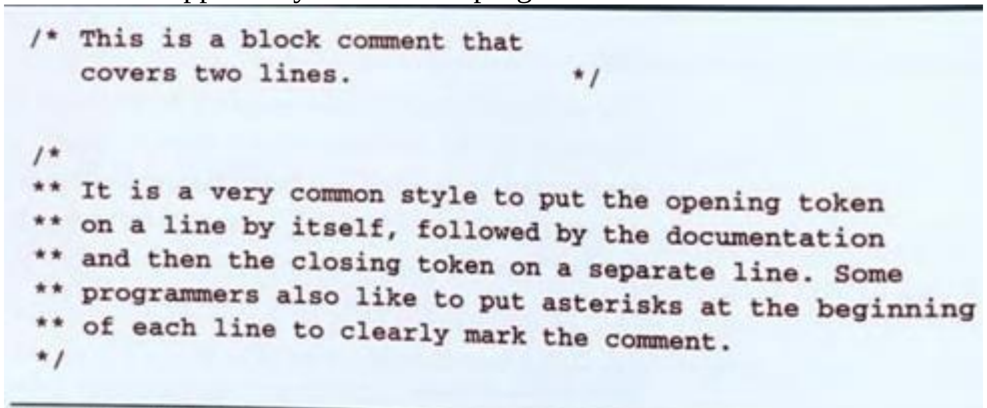
Two types of Comments:

- **Block comments (Multi line comments) and Line comments**.

  A block comment is used when comment spans for more than one line. Opening and closing tokens uses two characters. The opening token is /* and the closing token is */. Everything written inside these tokens are ignored by the compiler.
  /* This is a Block comment */

- A line comment uses two slashes(//) to identify a comment. This format does not require an end-of-comment token; Programmers generally use this format for short comments.
  //This is a single-line comment

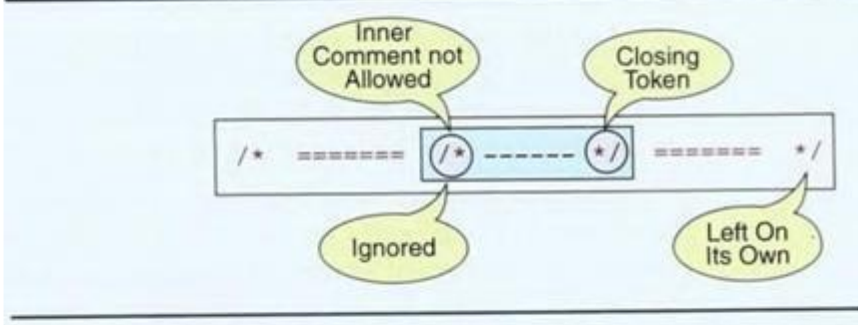
- These comments can appear anywhere in the program. Comments cannot be nested.



```
/* This is a block comment that
   covers two lines.            */

/*
** It is a very common style to put the opening token
** on a line by itself, followed by the documentation
** and then the closing token on a separate line. Some
** programmers also like to put asterisks at the beginning
** of each line to clearly mark the comment.
*/
```

Examples of Block Comments



Nested Block Comments Are Invalid

Program

/* The greeting program. This program demonstrates some of the components of a simple C program.

Written by: your name Date: date*/

```
#include<stdio.h>
int main( )
{
        // Local Declarations
        // Statements
        printf("Hello World\n");
        return 0;
}// main
```

SIMPLE ‘C’ PROGRAM:

```
/* simple program in c */
#include<stdio.h>
main()
{
printf(“welcome to c programming”);
} /* End of main */
```

Basics of a Computer Program- Algorithms, flowcharts (Using Dia Tool), pseudo code:

Algorithm:

A programming algorithm is a step by step procedure used for solving a problem.

Example:

Write an algorithm to find the largest of 2 numbers?

Algorithm:

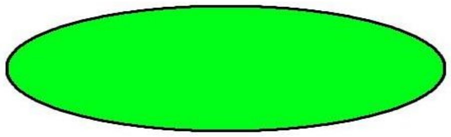
```
Step 1: Start
Step 2: Input the values of A, B
Step 3: Compare A and B.
Step 4: If A > B then go to step 6
Step 5: Print “B is largest” go to Step 7
Step 6: Print “A is largest”
Step 7: Stop
```

Flowchart: Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.

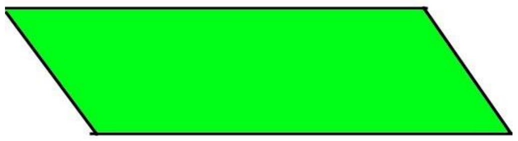
The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols used in Flowchart Designs:

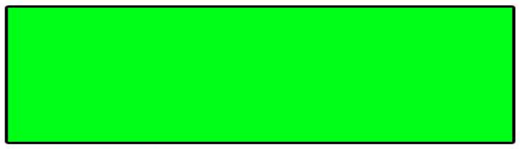
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



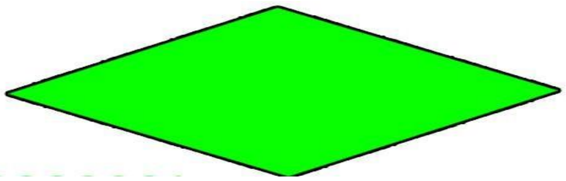
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



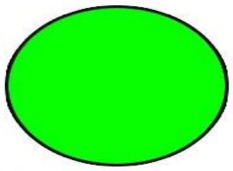
- **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.



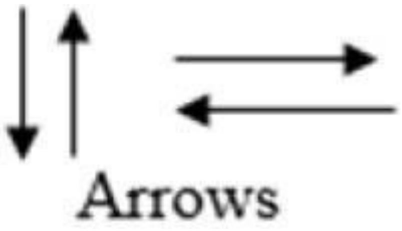
- **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



Rules For Creating Flowchart :
A flowchart is a graphical representation of an algorithm.it should follow some rules while creating a flowchart
Rule 1: Flowchart opening statement must be 'start' keyword.
Rule 2: Flowchart ending statement must be 'end' keyword.
Rule 3: All symbols in the flowchart must be connected with an arrow line.
Rule 4: The decision symbol in the flowchart is associated with the arrow line.

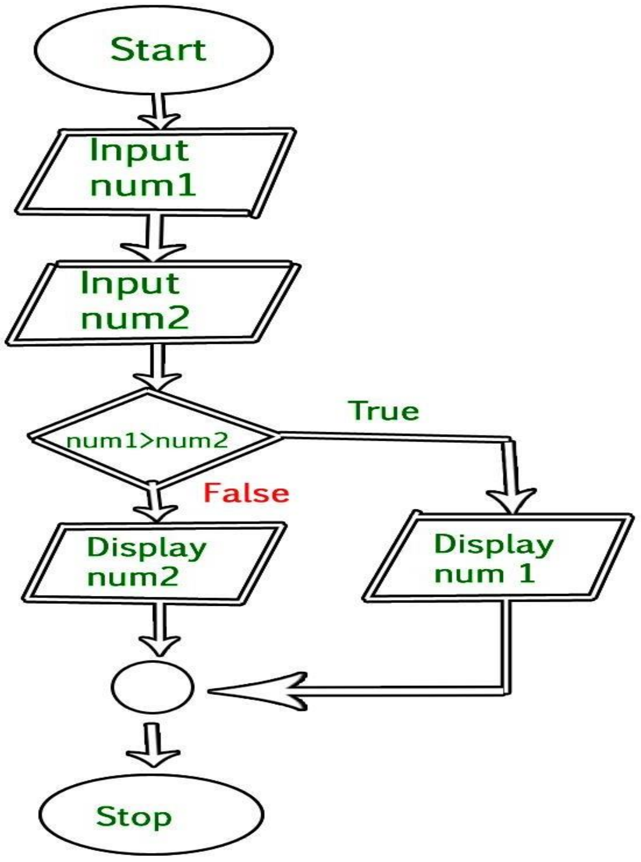
Advantages of Flowchart:
- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- It provides better documentation.

- Flowcharts serve as a good proper documentation.
- Easy to trace errors in the software.
- Easy to understand.
- The flowchart can be reused for inconvenience in the future.
- It helps to provide correct logic.

Disadvantages of Flowchart:

- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- Some developer thinks that it is waste of time.
- It makes software processes low.
- If changes are done in software, then the flowchart must be redrawn

  Example : Draw a flowchart to input two numbers from the user and display the largest of two numbers



<u>Pseudo code:</u>

The pseudo code in C is an informal way of writing a program for better human understanding. *Pseudo code* **is a method of representing logic in programming that is independent of any programming language.**

Let's consider an example of implementing a simple program that calculates the average of three numbers using pseudo code in C:

```
1. Start
2. Input three numbers
3. Calculate the sum of the three numbers
4. Divide the sum by 3 to get the average
5. Display the average
6. End
```

Pseudo code in C has the following Advantages :

1.Pseudo code is easier to read and comprehend than actual programming code since it describes program logic in everyday language.

***2.* Saving time:** By working out the specifics of the program logic before creating the actual code, producing pseudo code allows you to save time in the long run.

***3.*Reduces errors:** Because pseudo code enables you to develop the logic of your program in a more streamlined manner, it can help to decrease errors in your code.

4.Pseudo code is portable because it is ***language-independent*** and is simple to adapt to new programming languages.

<u>Creating and running programs/ Compilation and execution</u>

We have 4 steps to convert a program in high-level language into Machine language.

1. Writing and editing the program.
2. Compiling the program.
3. Linking the program with the required library modules and
4. Executing the program.

Writing and editing programs:

- The software used to write and edit programs is known as **text editor**.
- The text editor could be widely applicable is **word processor**, it is more special editor with **compiler** because these editors have some features like **copy, paste** of code from one place to another place in the program and can perform **alignthestatements**withtabs.
- After we complete the program **save the file** to the disk. This file will be given input to the compiler. This file is called **source file**.

Compiling Programs:

- The code in the source file to be converted into machine language, it is done by compiler.
- The C compiler is 2 separate programs. They are **preprocessor** and **translator**.
- Preprocessor reads the ***preprocessor commands***. These commands tell the preprocessor to look

for the special code libraries and make substitutions into the code. The result of preprocessing is called the **translation unit.**

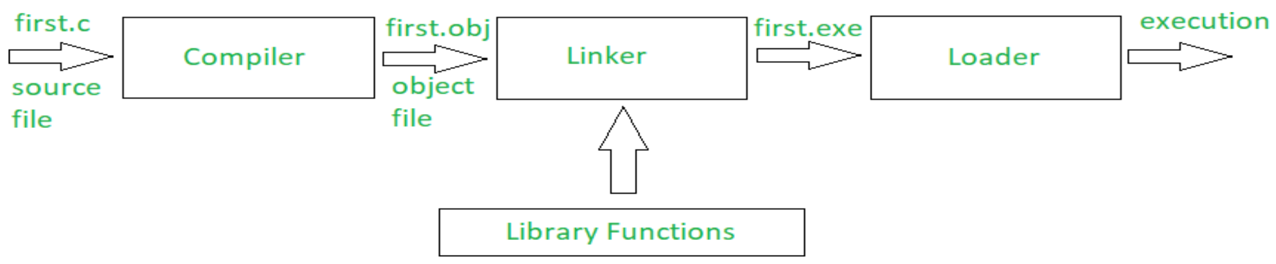
- After preprocessor has prepared the code the *translator* converts the code into machine language.
- The translator reads the translation unit and writes the resulting object module to a file. Even though the output of the compiler is machine language it is not completed to be executed.

Linking programs:

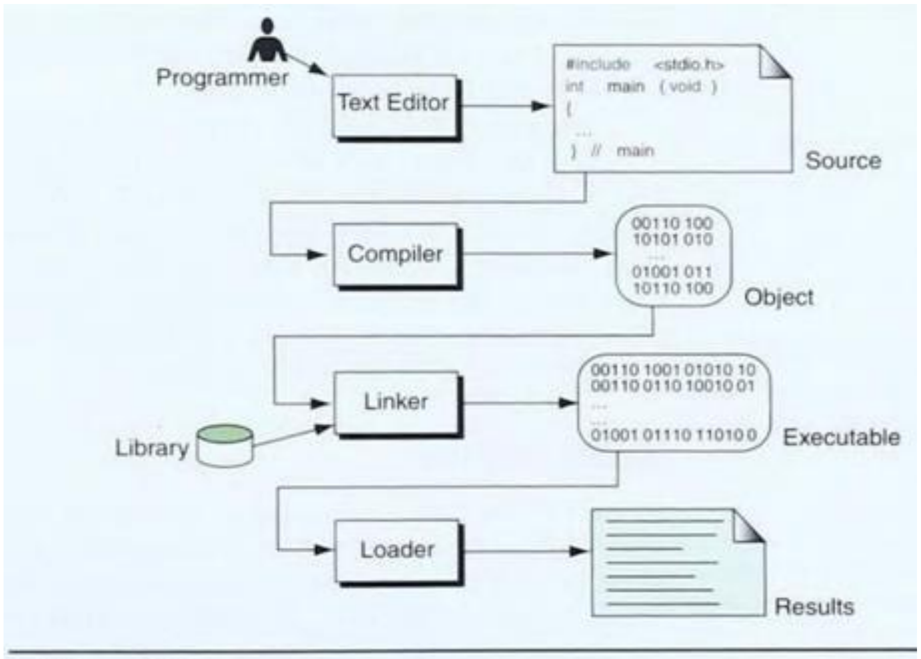
- Cprogram ismadeupofmanyfunctions. Thesefunctions are likeinput/outputprocesses, mathematical library functions, etc.
- The linker assembles all these functions into final executable program.

Executing Programs:

- Once our program is linked it is ready for execution. To execute a program, , we use Operating System command such as run to load the program into main memory..
- Getting the program into main memory is the function of the operating system program is known as loader. When the program is loaded execution takes place.
- In today's integrated environment all these processes are done in a single click or one pull down window.
- The program takes the input from a file or from the user. After processing/ execution is done output is displayedonthemonitororstoredinafile.
- When the program has finished its job, it tells the operating system which then removes the program from the memory.



-



Building a C Program

Identifiers

Names of the variables and other program elements such as functions, array, etc, are known as identifiers. There are few rules that govern the way variable are named (identifiers).
Identifiers can be named from the combination of A-Z, a-z, 0-9, _ (Underscore).

- The first alphabet of the identifier should be either an alphabet or an underscore. Digitsare not allowed.
- First 63 characters of an identifier are significant.
- It may not have a space or a hyphen (-). It may not have a special characters like $,@,% ,*, etc.
- It should not be a keyword.Eg:
  name, ptr, sum [valid]
  Eg: sum-sal, 2name, std no, int [invalid]

Valid names	**Invalid names**
a // valid but poor style	$sum //$ is illegal
student_name	2names // first char digit
_astudentName	aum-salary // contains hyphen
_Bool // Boolean system id	stdnt Nmbr // contains spaces
INT_MIN // System Defined Value	int // Keyword

Keywords

Definition:

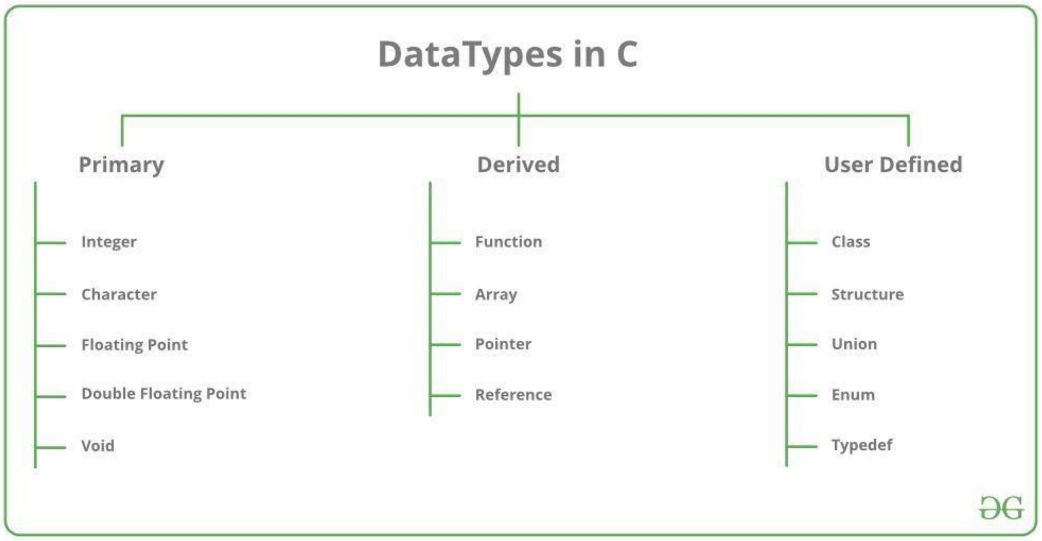
Keywords are the system defined identifiers (reserved identifiers) and cannot be as the names of the program variables or other user defined program elements (user defined function names).

1. All keywords have fixed meanings that do not change.
2. White spaces are not allowed in keywords.
3. Keyword may not be used as an identifier.
4. It is strongly recommended that keywords should be in lower case letters.
5. There are totally 32(Thirty-Two) keywords used in a C programming.

int	float	double	Long	short	signed	unsigned	const
if	Else	switch	break	default	do	While	for
register	extern	static	struct	typedef	enum	return	sizeof
goto	union	auto	Case	void	char	continue	volatile

Data Types:

- Atype defines a **setofvalues** and a **setofoperations** that can be applied onthose values.
- The data type specifies the size and type of information the variable will store.



Primitive data types/Primary datatypes in C:

1.Integer Data Type:

The integer datatype in C is used to store the integer numbers(any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can be stored in int data type in C.
The storage size of int data type is 2 or 4 or 8 byte.

It varies depend upon the processor in the cpu that we use.

If we are using 16 bitprocessor, 2 byte(16 bit) of memory will be allocated for int data type.

like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memoryfor 64 bit processor is allocated for int datatype.

int (2 byte) can store values from -32,768 to +32,767

int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.

- **Range:** -2,147,483,648 to 2,147,483,647
- **Size:** 4 bytes
- **Format Specifier:** %d

Syntax of Integer
We use **int keyword** to declare the integer variable:

int *var_name;*

The integer data type can also be used as

1. **unsigned int:** Unsigned int data type in C is used to store the data values from zero to positive numbers but it can't store negative values like signed int.
2. **short int:** It is lesser in size than the int by 2 bytes so can only store values from -32,768 to 32,767.
3. **long int:** Larger version of the int datatype so can store values greater than int.
4. **unsigned short int:** Similar in relationship with short int as unsigned int with int.

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

    int a = 9;   // Integer value with positive data.

    int b = -9;    // integer value with negative data.

    int c = 89U;      // U or u is Used for Unsigned int in C.

    long int d = 99998L; // L or l is used for long int in C.
    printf("Integer value with positive data: %d\n", a);
    printf("Integer value with negative data: %d\n", b);
    printf("Integer value with an unsigned int data: %u\n", c);
    printf("Integer value with an long int data: %ld", d);
    return 0;
}
```

Output:

```
Integer value with positive data: 9

Integer value with negative data: -9

Integer value with an unsigned int data: 89

Integer value with an long int data: 99998
```

2. Character Data Type

Character data type allows its variable to store only a single character. The size of the character is 1 byte.

- **Range:** (-128 to 127) or (0 to 255)
- **Size:** 1 byte
- **Format Specifier:** %c

Syntax of char

The **char keyword** is used to declare the variable of character type:

char *var_name;*

program:

```
#include <stdio.h>
int main()
{
    char a = 'a';
    char c ;

    printf("Value of a: %c\n", a);

    a++;
    printf("Value of a after increment is: %c\n", a);

    /*c is assigned ASCII value which corresponds to the character 'c' , a-->97 b-->98 c-->99 here c will
be printed.*/
    c = 99;

    printf("Value of c: %c", c);

    return 0;
}
```

Output

```
Value of a: a

Value of a after increment is: b

Value of c: c
```

3.Float Data Type

In C programming float data type is used to store floating-point values. Float in C is used to store decimal and exponential values. It is used to store decimal numbers (numbers with floating point values) with single precision.

- **Range:** 1.2E-38 to 3.4E+38
- **Size:** 4 bytes
- **Format Specifier:** %f

Syntax of float

The **float keyword** is used to declare the variable as a floating point:

float *var_name;*

program:

```
#include <stdio.h>

int main()
{
   float a = 9.0f;
   float b = 2.5f;

   //   2x10^-4
   float c = 2E-4f;
   printf("%f\n", a);
   printf("%f\n", b);
   printf("%f", c);
   return 0;
}
```

Output

```
9.000000

2.500000

0.000200
```

4.Double Data Type

A Double data type in C is used to store decimal numbers (numbers with floating point values) with double precision.

It is used to define numeric values which hold numbers with decimal values in C.

- **Range:** 1.7E-308 to 1.7E+308
- **Size:** 8 bytes
- **Format Specifier:** %lf

Syntax of Double:

The variable can be declared as double precision floating point using the **double keyword:**

double *var_name;*

program:

```
#include <stdio.h>

int main()
{
   double a = 123123123.00;
```

```
    double b = 12.293123;
    double c = 2312312312.123123;

    printf("%lf\n", a);

    printf("%lf\n", b);

    printf("%lf", c);

    return 0;
}
```

Output:

```
123123123.000000

12.293123

2312312312.123123
```

Type	Storage size	Format specifier	Value range	Precision
float	4 bytes	%f	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	%lf	2.3E-308 to 1.7E+308	15 decimal places
long double	10 bytes	%Lf	3.4E-4932 to 1.1E+4932	19 decimal places

Data Type	Size (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	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long long int	8	-(2^63) to (2^63)-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	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	16	3.4E-4932 to 1.1E+4932	%Lf

5.Void Data Type

- The void type, used with the keyword void, has **no values** and **no operations**.
- Void is an empty data type that has no value.
- This can be used in functions and pointers.

Variables

- Variables are containers for storing data values, like numbers and characters.
- Variable is a name given to memory location that holds a value.
- **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

Rules for defining a 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 names:	*invalid names*:
int a;	**int** 2;
int _ab;	**int** var name;
int a30;	**int** long;

Variable decleration:

Syntax to declare a variable:

```
datatype  variable_list;
```

Eg: **int** a;
float b;
char c;

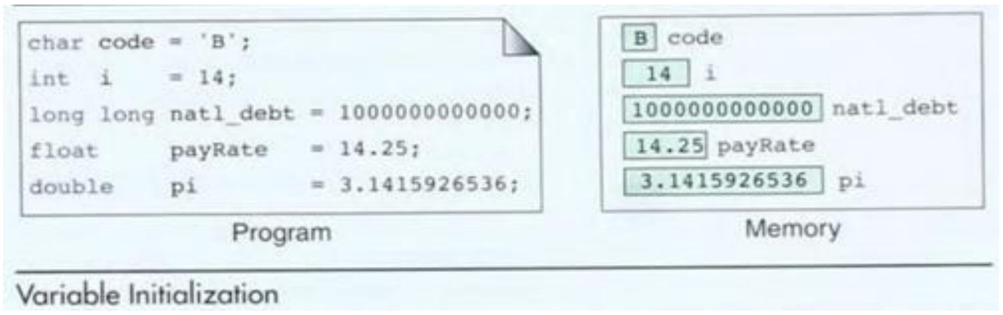
Here, a, b, c are variables. The int, float, char are the data types.

Variable initialization

Datatype variable_name= value;

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;
char c='A';
```



```
char code = 'B';
int  i    = 14;
long long natl_debt = 1000000000000;
float     payRate   = 14.25;
double    pi        = 3.1415926536;
```

Variable Initialization

Types of variables: 1.Local variables 2.Global variables
Local Variables:
A variable that is declared inside the function or block is called a local variable.

- The scope of the local variable is with in the fuctions only.
- Local variables can't be accessed outside the function.
- By default value of a local variable is Garbage value.

```
//program for local variables
#include <stdio.h>
int main()
 {
   int a=10;        //local variables a and b
   float b;
   printf("%d\n%f\n",a,b);
   return 0;
}
```

Global Variables:
A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.
The scope of the global variable is through out a program.

- The default value of a global variable is zero.

```
//program for global variables
#include <stdio.h>
int  c=20;  //global  variable
int b;          //global variable
int main()
 {
   int   a=10;   //local   variables   a
   printf("%d\n%d\n%d\n",a,b,c);
   return 0;
}
```

Difference between identifier and variable

- The **identifier** is only used to identify an entity uniquely in a program at the time of execution whereas, a **variable** is a name given to a memory location that is used to hold a value.
- **Variable** is only a kind of **identifier**, other kinds of **identifiers** are function names, class names, structure names, etc.

Constants

- Constants are data values that cannot be changed during the execution of a program. Like variables, constants have a type.

Types of constants:

1. Numeric constants

a) Integer constants:

Decimal,Octal,Hexadecimal

b)Real constants

2. Character constants

Single character constants

String constants

1. Integer constants:

Decimal Constants:

A whole number represented in ***base 10*** is known as a ***decimal constant***. It has digits that range from ***0*** to ***9***. Declaring a ***decimal constant*** has a simple syntax that just requires the value to be written.

Example:

```
#include <stdio.h>

int main()
 {
int decimal = 42;
printf("The decimal constant is: %d\n", decimal);
return 0;
}
```

Output:

```
The decimal constant is: 42
```

Octal Constant:

A ***base 8*** value is represented by an ***octal constant***. It is prefixed with a ***'0' (zero)*** to show that it is an octal constant and has digits ranging from ***0*** to ***7***.

Example:

```
#include <stdio.h>

int main()
 {
int octal = 052; // Octal representation of decimal 42
printf("The octal constant is: %o\n", octal);
return 0;
}
```

Output:

```
The octal constant is: 52
```

Hexadecimal Constant:

A ***base-16*** value is represented by a ***hexadecimal constant***. It uses letters ***A to F*** (or ***a to f***) and numbers ***0*** to ***9*** to represent values from ***10*** to ***15***. It is prefixed with ***'0x'*** or ***'0X'*** to identify it as a hexadecimal constant.

Example:

```
#include <stdio.h>

int main()
{
int hexadecimal = 0x2A;       // Hexadecimal representation of decimal 42
printf("The hexadecimal constant is: %x\n", hexadecimal);
return 0;
}
```

Output:

```
The hexadecimal constant is: 2a
```

2. Real or Floating-Point Constant:

A ***fractional component*** or ***exponentiation*** of a number is represented by ***a real or floating-point constant***. It can be expressed with a decimal point, the letter ***"E"***, or the symbol ***"e"*** in exponential or decimal notation.

Example:

```
#include <stdio.h>

int main()
 {
float real = 3.14;
printf("The real constant is: %f\n", real);
return 0;
}
```

Output:

```
The real constant is: 3.140000
```

2. Character Constant:

Single character constant:

A ***character constant*** represents a ***single character*** that is enclosed in ***single quotes***.

Example: **char c='a';**

char c='5';

ASCII character	Symbolic name	ASCII character	Symbolic name
Null character	**'\0'**	Form feed	**'\f'**
Alert (bell)	**'\a'**	Carriage return	**'\r'**
Backspace	**'\b'**	Single quote	**'\''**
Horizontal tab	**'\t'**	Double quote	**'\"'**
New line	**'\n'**	Backslash	**'\\'**
Vertical tab	**'\v'**		

Example:

```
#include <stdio.h>

int main()
 {
char character = 'A';
printf("The character constant is: %c\n", character);
return 0;
}
```

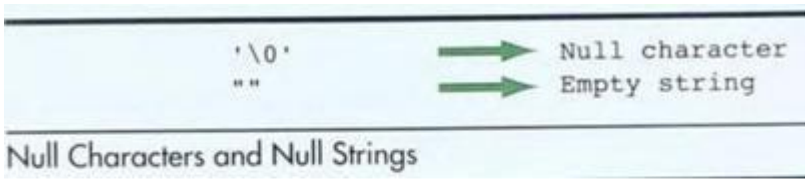
Output:

```
The character constant is: A
```

String Constant:

A ***sequence of zero or more characters*** wrapped in ***double quotes*** is represented by a ***string constant***. It is a character array that ends with the null character \0.

Example:
" " // a null string
"h"
"computer"
"wide characters"



Null Characters and Null Strings

Example:

```
#include <stdio.h>

int main()
 {
char string[] = "Hello, World!";
printf("The string constant is: %s\n", string);
return 0;
}
```

Output:

```
The string constant is: Hello, World!
```

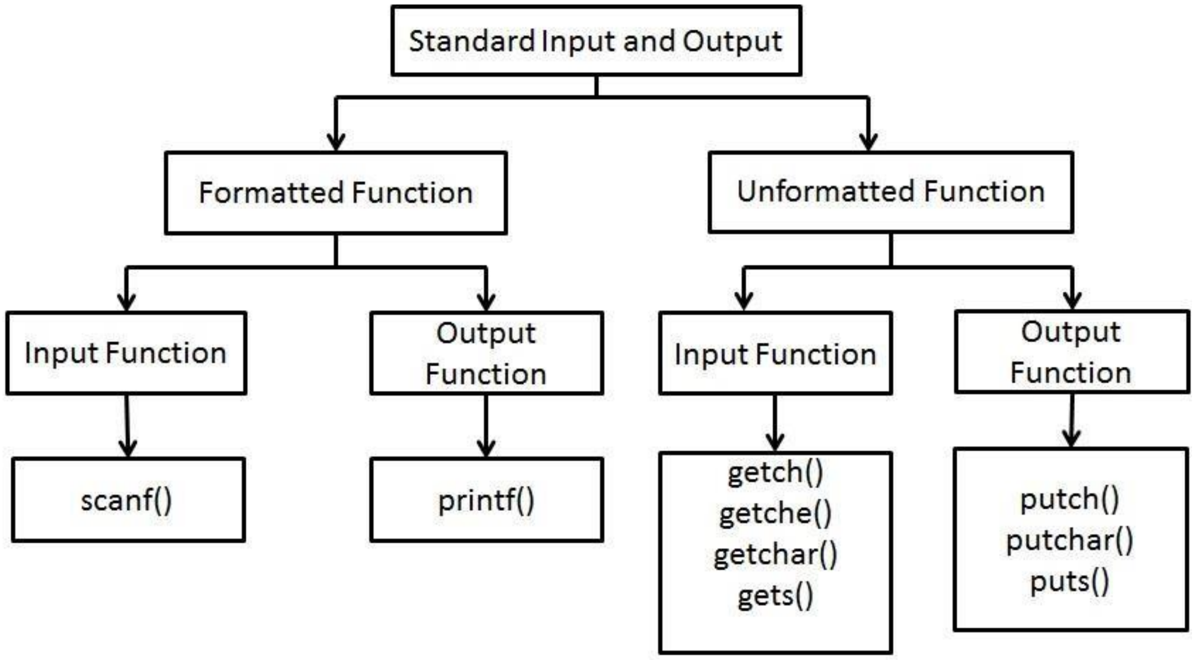
Basic Input/output

Standard input-- used to take the input from the device such as Keyboard.

Standard output--used to send output to a device such as monitor or screen.

Types of Input Output:

There are 2 kinds of I/O functions as given below.



Unformatted I/O Functions

getchar() :

The ***getchar***() reads character type data from input file ***stdin***. It reads one character at a time till the user pressses the enter key. The ***getchar***() returns a character type. It reads next character in the input file.

Syntax:

```
        Variablename= getchar();
```

putchar() :

The ***putchar***() writes the character type data to the output file ***stdout***(screen). It is used to print one character at a time.

Syntax:

```
        Putchar(Variable_name);
```

//getchar putchar example:

```
#include<stdio.h>
int main()
{
  char a;
  printf("enter a character \n");
  a = getchar();
  printf("you entered character is \t");
  putchar(a);
  return 0;
}
```

Output:

```
enter a character
A
you entered character is   A
```

gets()

The ***gets***() is used to read string from standard input file ***stdin*** and places it into some buffer which the programmer must provide. The ***gets***() stops reading when an enter key is pressed or EOF is reached.

Syntax:

```
        gets(variablename);
```

puts() :

The *puts*() is used to send the string to the output file *stdout*, until it finds a NULL end of string marker. The NULL is not written, instead the newline character is written to the output *stdout*.

Syntax:

```
        puts(variablename);
```

```
//gets puts example
#include<stdio.h>
int main()
{
  char ch[20];
  printf("Enter the text: ");
  gets(ch);
  puts(ch);
  return(0);
}
```

Output:

```
Enter the text: welcome
welcome
```

Formatted Input Output Functions:

1. .scanf():

- The ***scanf***() reads all types of data values given by the user and these values are assigned to the variables. It requires the conversion specification (such as %s and %d) to identify the type of data to be read during the program execution.
- You need to pass the address of the variable to the ***scanf*** function along with other arguments so that the read values can be assigned to the correct destination.

Syntax: **scanf("control string", address list);**
(or)
scanf("format specifiers", &variable1,&variable2...);

Example:

```
int x;
float y;
scanf ("%d % f ", &x, &y);
```

Control string

- It tells the function the type of data being Input and uses the same control sequences as the printf() function**.**
- It enclosed with double quotes (" ").
- It specifies the type of values that have to be read from keyboard. Control string consists of format specification.

Example:

Integer - %d Float - %f double - %lf long double - %Lf char - %c

Addresses list

- It contain addresses of i/o variables proceed the addresses are specified by before the variable by ampersand (&) operator.

scanf() reads the characters from standard input and interprets them according to the format string specification. It stops reading when it encounters a space or when some input fails to match with the conversion specification.

2 .printf() :

- The ***printf***() prints all types of data value to the standard output(typically the screen). It requires a conversion symbol and variable name to print the data. The ***printf***() is part of every C compiler. Its header file is stdio.h.

Syntax:

printf("Format control string", data_value1, data_value2, . . . , data_value$_n$);

The first argument is the format string which is composed of ordinary characters and conversion specification. This is followed by zero or more optional arguments.

Control String

1. Format Specifier given in the form of % specifier.
2. Escape sequence characters such as \n (new line) \t(tab).
3. It can have a string that must be printed on the monitor.

- The format of numbers displayed by printf() can be controlled by **field with specifiers** included as part of each conversion control sequence.

printf("sum of %.3d and %4d is %5d",6,15,21);

Output: **sum of □□6 and □□□5 is □□□21**

Example program using *scanf*() and *printf*() functions:

```
#include<stdio.h>
int main()
{
  int a;
  printf("Enter the value of a = ");
  scanf("%d",&a);
  printf("\nThe value of a = %d",a);
  return(0);
}
```

Output:

```
Enter the value of a = 10

The value of a = 10
```

//Example program using field width in print statement

```
#include<stdio.h>
int main()
{
        printf("\n%3d",6);
        printf("\n%3d",18);
        printf("\n%3d",124);
        printf("\n----");
        printf("\n%9d",6+18+124);
        return 0;
}
```

Output:

```
6
 18
124
---
    148
```

Float point

%5.2f – 42.30

%5.2f – 142.36

%10.3f – AAAA25.670

Format Modifier (modifiers for printf formats)

- A format specification can also include “modifiers” that can control how much of the item’s values is printed and how much space it gets.
- The modifiers come between the % and the format control letter.

Justification

- It controls the placement of a value when it is shorter than the specified width.
- Justification can be left or right. The format modifier (-) minus is used.

Left Justification

- A format specification has (-) minus sign used before the width modifier says to left justify the argument within its specified width.

Example: **printf(“%-10d”,59);**

Output: **59□□□□□□□□□**

Right Justification + format modifier (or) explicit sign display

- The positive sign is used before the width modifier, says to right justify the argument with its specified width.

Example: **printf("%+10d",59);**

Output: □□□□□□□□**59**

printf() using left and right justification

```
#include<stdio.h>
int main()
{
        printf("%+10s\n","right");
        printf("%-10s\n","left");
        printf("%+10s\n","abcd");
        printf("%-10s\n","abcd");
        return 0;
}
```

Output:

```
right
left
      abcd
abcd
```

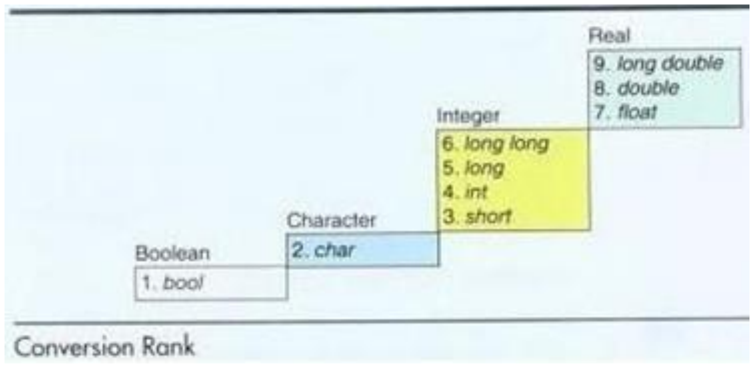
Usually the format string is enclosed with the double quotation marks in both ***scanf***() and ***printf***().

Type Conversion

- Type converse on can be defined as **converting one data type** into **another data type**.
- There are 2 different type conversions are there they are:
  1. Automatic or **implicit** type conversion.
  2. Type casting **or explicit** type conversion.

Implicit Type Conversion

- When the type conversion is performed automatically by the compiler without programmer intervention such type of conversion is known as implicit type conversion.
- If operators are different types the **lower type** is automatically converted to **higher type** before operation proceed.
- The following Figure shows the ranks as we use them for conversion.



- There are two types of conversions in implicit type conversion as:
    1. Assignment Type Conversion
    2. Automatic Type Conversion

Assignment Type Conversion

- This means that the value of the expression on the right side of the equal sign is converted to the data type of the variable to the left of the equal sign.

Example: **result = 4;**

- When 'result' is declared as double '4' is an integer then '4' will be converted to the double Precision value 4.0 and the value is assigned to the result.

Automatic Type Conversion

- The data type have lower rank is converted automatically in to higher rank data before the operation process.
- This automatic conversion across an assignment operator is referred to as integer type conversion

Example program for implicit type conversion

```
#include <stdio.h>
 int main ()
{
        double a = 2.5,s;
        int b=2;
        s=a+b;
        printf("value of s = %d “, s);
}
```

Output:

```
value of s = 4
```

Explicit Type Conversion/Type Casting

- Type conversion performed by the programmer to change the data type of the expression of a specific type is known as explicit type conversion.

Syntax: **(data type) expression**

Example: **double sum/count;**

Example program for explicit type conversion

```
#include <stdio.h>
 int main ()
{
        int sum=17,count=5;
        double mean;
        mean=(double)sum/count;
        printf("value of mean = %f “, mean);
}
```

Output:

```
value of mean = 3.400000
```

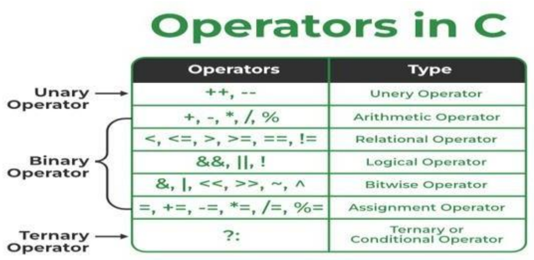
Operators:

Operators can be defined as the symbols that help us to perform specific mathematical, relational, bitwise, conditional, or logical computations on operands. In other words, we can say that an operator operates the operands.

For example, ‘+’ is an operator used for addition, as shown below:

c = a + b;

Here, ‘+’ is the operator known as the addition operator, and ‘a’ and ‘b’ are operands. The addition operator tells the compiler to add both of the operands ‘a’ and ‘b’. The functionality of the C programming language is incomplete without the use of operators.


Operators in C

	Operators	Type
Unary Operator	++, --	Unery Operator
Binary Operator	+, -, *, /, %	Arithmetic Operator
	<, <=, >, >=, ==, !=	Relational Operator
	&&, ||, !	Logical Operator
	&, |, <<, >>, ~, ^	Bitwise Operator
	=, +=, -=, *=, /=, %=	Assignment Operator
Ternary Operator	?:	Ternary or Conditional Operator

Types of Operators in C:

C operators can be classified into the following types:

1. Arithmetic operators
2. Increment and decrement operators
3. Relational operators
4. Logical operators

5. Bitwise operators
6. Assignment operators
7. Conditional operators
8. Special operators

1. Arithmetic Operators :

The C language supports all the basic arithmetic operators such as **addition**, **subtraction**, **multiplication**, **division**, etc.

Operator	Description	Example(where a and b are variables with some integer value)
+	adds two operands (values)	a+b
-	subtract second operands from first	a-b
*	multiply two operands	a*b
/	divide the numerator by the denominator, i.e. divide the operand on the left side with the operand on the right side	a/b
%	This is the **modulus operato**r, it returns the remainder of the division of two operands as the result	a%b
++	This is the **Increment operator** - increases the integer value by one. This operator needs only a **single operand**.	a++ or ++a
--	This is the **Decrement operator** - decreases integer value by one. This operator needs only a **single operand**.	--b or b--

Program using arithmetic operators:

```
int main()
{
    int a = 50, b = 23, result;
    // addition
    result = a+b;
    printf("Addition of a and b = %d \n",result);
    // subtraction
    result = a-b;
    printf("Subtraction of a  and b = %d \n",result);
    // multiplication
    result = a*b;
    printf("Multiplication of a and b = %d \n",result);
    // division
    result = a/b;
    printf("Division of a and b = %d \n",result);
  //modulus
   result = a%b;
   printf("modulus of a and b = %d",result);
    return 0;
}
```

Output:

```
Addition of a and b = 73
Subtraction of a and b = 27
Multiplication of a and b = 1150
Division of a and b = 2
modulus of a and b = 4
```

2. Increment and decrement operators:

- The increment operator is used to increase the value of any numeric value by 1.
- The decrement operator is used to decrease the value of any numeric value by 1.

++ increment operator, -- decrement operator

- When we use the increment and decrement operator as a **prefix** (means before the operand), then, first the increment operation is done and that value is used, like in the first two printf() functions, we get the updated values of a and b.
- Whereas when we use the increment and decrement operators as **postfix** (means after the operand), then, first the larger expression is evaluated which is printf() in this case and then the value of the operand is updated.

Program for using increment and decrement operators:

```
#include <stdio.h>

int main()
{

  int a = 10, b = 20, c, d;

  /*      Using increment operator  */
  printf("Incrementing value of a = %d \n", ++a);

  /*      Using decrement operator  */
  printf("Decrementing value of b = %d \n", --b);

  // first print value of a, then decrement a
  printf("Decrementing value of a = %d \n", a--);
  printf("Value of a = %d \n", a);

  // first print value of b, then increment b
  printf("Incrementing value of b = %d \n", b++);
  printf("Value of b = %d \n", b);

  return 0;

}
```

Output:

```
Incrementing value of a = 11
Decrementing value of b = 19
Decrementing value of a = 11
Value of a = 10
Incrementing value of b = 19
Value of b = 20
```

3. Relational operators in C

- The relational operators (or **comparison** operators) are used to check the relationship between two operands.
- It checks whether two operands are **equal** or **not equal** or **less than** or **greater than**, etc.
- It returns **1** if the relationship checks **pass**, otherwise, it returns **0**.
- When we use relational operators, we get the output based on the result of the comparison.
- If it's **true**, then the output is **1** and if it's **false**, then the output is **0**.

Operator	**Description**	**Example** (a and b, where a = 10 and b = 11)
==	Check if the two operands are equal	a == b, returns 0
!=	Check if the two operands are not equal.	a != b, returns 1 because a is not equal to b
>	Check if the operand on the left is greater than the operand on the right	a > b, returns 0
<	Check operand on the left is smaller than the right operand	a < b, returns 1
>=	check left operand is greater than or equal to the right operand	a >= b, returns 0
<=	Check if the operand on the left is smaller than or equal to the right operand	a <= b, returns 1

Program for relational operators:

```
#include <stdio.h>
int main()
{
   int a = 10, b = 20, result;

 // Equal
  result = (a==b);
 printf("(a == b) = %d \n",result);

 // less than
```

```
    result = (a<b);
    printf("(a < b) = %d \n",result);

    // greater than
    result = (a>b);
    printf("(a > b) = %d \n",result);

    // less than equal to
    result = (a<=b);
    printf("(a <= b) = %d \n",result);

    return 0;

}
```

Output:

```
(a == b) = 0
(a < b) = 1
(a > b) = 0
(a <= b) = 1
```

4.Logical Operators :

C language supports the following 3 logical operators.

Operator	Description	Example (a and b, where a = 1 and b = 0)
&&	**Logical AND**	**a && b, returns 0**
||	**Logical OR**	**a || b, returns 1**
!	**Logical NOT**	**!a, returns 0**

These operators are used to perform logical operations and are used with conditional statements like if else.

1. With the **AND** operator, only **if both operands are true**, the **result is true**.
2. With the **OR** operator, if a **single operand is true**, then the **result will be true**.
3. The **NOT** operator **changes true to false**, and **false to true**.

Program for logical operators:

```
#include <stdio.h>

int main() {

  int a = 1, b = 0, result;

  // And
  result = (a && b);
  printf("a && b = %d \n",result);

  // Or
  result = (a || b);
  printf("a || b = %d \n",result);

  // Not
  result = !a;
  printf("!a = %d \n",result);

  return 0;

}
```

Output:

```
a && b = 0

a || b = 1

!a = 0
```

5.Bitwise operators:

- Bitwise operators perform manipulations of data at the bit level.
- These operators also perform the **shifting of bits from right to left**.
- Bitwise operators are not applied to float or double, long double, void, etc.
- There are **6 bitwise operators** in C programming.

The following table contains the bitwise operators.

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	One's complement
>>	Shift Right
<<	Shift Left

The bitwise AND, OR, and NOT operator works the same way as the Logical AND, OR, and NOT operators, except that the bitwise operators work **bit by bit**.

we have a **truth table** for showing how these operators work with different values.

a	**b**	**a & b**	**a | b**	**a ^ b**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise operators can produce any arbitrary value as a result. It is not mandatory that the result will either be 0 or 1.

Bitwise >> and << operators

- The bitwise shift operator **shifts the bit value**, either to the left or right.
- The **left operand** specifies the **value to be shifted** and the **right operand** specifies the **number of positions** that the bits in the value have to be shifted.
- Both operands have the same precedence.

Understand, how bits shift from left to right and vice versa.

a = 00010000

b = 2

a << b = 01000000

a >> b = 00000100

In case of a << b, **2 bits** are shifted to left in **00010000** and additional zeros are added to the opposite end, that is right, hence the value becomes **01000000**

And for a >> b, **2 bits** are shifted from the right, hence two zeros are removed from the right and two are added on the left, hence the value becomes **00000100**

Please note, shift doesn't work like rotating, which means, the bits shifted are not added at the other end. The **bits that are shifted are lost**.

Program for bitwise operators:

```
#include <stdio.h>
int main()
{
 int a = 0001000, b = 2, result;
  // <<
  result = a<<b;
  printf("a << b = %d \n",result);

  // >>
  result = a>>b;
  printf("a >> b = %d \n",result);
  return 0;
}
```

Output:

```
a << b = 2048
a >> b = 128
```

6. Assignment Operators:

- The assignment operators are used to assign value to a variable.
- For example, if we want to assign a value **10** to a variable x then we can do this by using the assignment operator like: x = 10; Here, = (equal to) operator is used to assign the value.
- In the C language, the = (equal to) operator is **used for assignment** however it has several other variants such as +=, -= to combine two operations in a single statement.

Operator	Description	Example (a and b are two variables, with where a=10 and b=5)
=	assigns values from right side operand to left side operand	a=b, a gets value 5
+=	adds right operand to the left operand and assign the result to left operand	a+=b, is same as a=a+b, value of a becomes 15
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b, is same as a=a-b, value of a becomes 5

Operator	Description	Example (a and b are two variables, with where a=10 and b=5)
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b, is same as a=a*b, value of a becomes 50
/=	divides left operand with the right operand and assign the result to left operand	a/=b, is same as a=a/b, value of a becomes 2
%=	calculate modulus using two operands and assign the result to left operand	a%=b, is same as a=a%b, value of a becomes 0

- When we combine the arithmetic operators with the assignment operator =, then we get the **shorthand form** of all the arthmetic operators.

Example program Using Assignment Operators:

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

  int a = 10,b=30,c=10;
  // += operator
  a += 20;
  printf("result = %d \n",a);

  // -= operator
  b -= 5;
  printf("result = %d \n",b);

  // *= operator
  c *= 10;
  printf("result = %d \n",c);

  return 0;
}
```

Output:

```
result = 30
result = 25
result = 100
```

7. Ternary Operator (?)/Conditional operator:

The ternary operator, also known as the conditional operator in the C language can be used for statements of the form if-then-else.

The **basic syntax** for using ternary operator is:

(Expression1)? Expression2 : Expression3;

Here is how it works:

- The question mark ? in the syntax represents the if part.
- The first expression (expression 1) returns either **true** or **false**, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns **true** then the (expression 2) is executed.
- If (expression 1) returns **false** then the expression on the right side of : i.e (expression 3) is executed.

Example Program using ternary operators:

```
 /* Using ternary operator
   - If a == b then store a+b in result
   - otherwise store a-b in result
 */

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

  int a = 20, b = 20, result;
  result = (a==b)?(a+b):(a-b);

  printf("result = %d",result);
  return 0;
}
```

Output:

```
result = 40
```

8. **Special Operator s:** &, *, sizeof, etc:

sizeof operator is used to find size of any entity(variable, array, etc.), & operator to find the address of a variable, etc. You can see a list of such operators in the below table.

.

Operator	**Description**	**Example**
sizeof	returns the size(length in bytes) of the entity, for eg. a variable or an array, etc.	sizeof(x) will return the size of the variable x
&	returns the memory address of the variable	&x will return the address of the variable x
*	represents a pointer to an object. The * operator returns the value stored at a memory address.	m = &x (memory address of the variable x) *m will return the value stored at the memory address m
. (dot) operator	used to access individual elements of a C structure or C union.	If emp is a structure with an element int age in it, then emp.age will return the value of age.
-> (arrow) operator	used to access structure or union elements using a pointer to the structure or union.	If p is a pointer to the emp structure, then we can access age element using p->age
[] operator	used to access array elements using indexing	if arr is an array, then we can access its values using arr[index], where index represents the array index starting from zero

Program using size of and & operator:

```
#include <stdio.h>
int main()
{
  int a = 20;
  char b = 'B';
  double c = 17349494.249324;

  // sizeof operator
  printf("Size of a is: %d \n", sizeof(a));
  printf("Size of b is: %d \n", sizeof(b));
  printf("Size of c is: %d \n", sizeof(c));
  // & operator
  printf("Memory address of a: %d \n", &a);
  return 0;
}
```

Output:

```
Size of a is: 4
Size of b is: 1
Size of c is: 8
Memory address of a: -1254020756
```