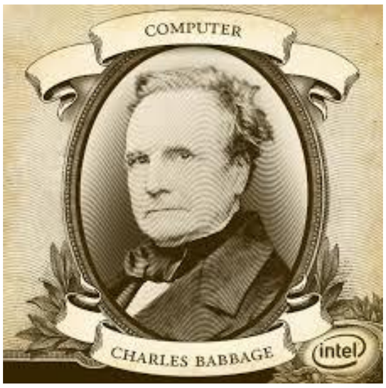
UNIT-I

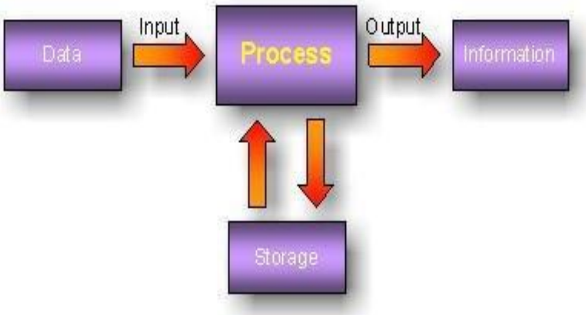
Introduction to Computer:

A Computer is an electronic device that takes data and instructions as an ***input*** from the user, ***process*** data, and provides useful information known as ***output***. This cycle of operation of a computer is known as the input-process-output cycle.



Charles Babbage (26 December 1791 – 18 October 1871) was an English polymath.

He is known as father of computers.
Invented first analytical computer in year 1822.



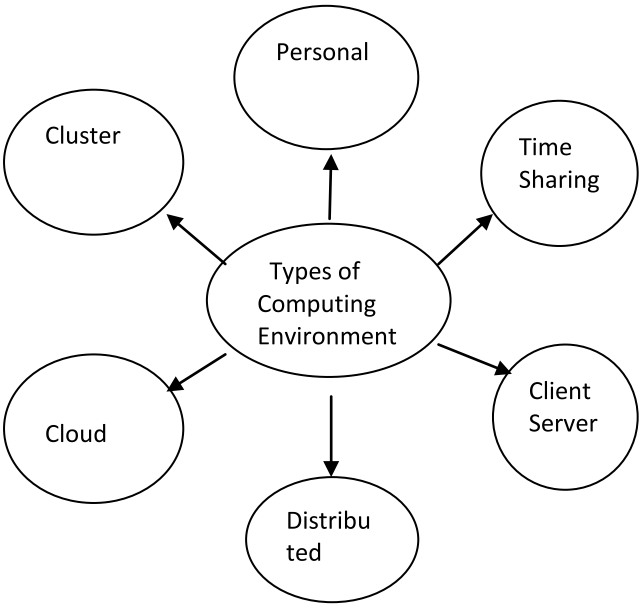
- ✓ **Data :** Data is the collection of different types of entities (i.e A-Z,0-9, symbols and character sets etc.) or the raw facts of a computers are called data.
- ✓ **Information :** The meaning full results occurred from data after performing process on it.
- ✓ **Process :** The intermediate work performed between data and information is know as process.

1.1 Computing Environments :

Computing Environment is a collection of computers / machines, software, and networks that support the processing and exchange of electronic information meant to support various types of computing solutions.

Types of Computing Environments

- Personal Computing Environment.
- Time sharing Environment.
- Client Server Computing Environment.
- Distributed Computing Environment.
- Cloud computing Environment.
- Cluster computing Environment.

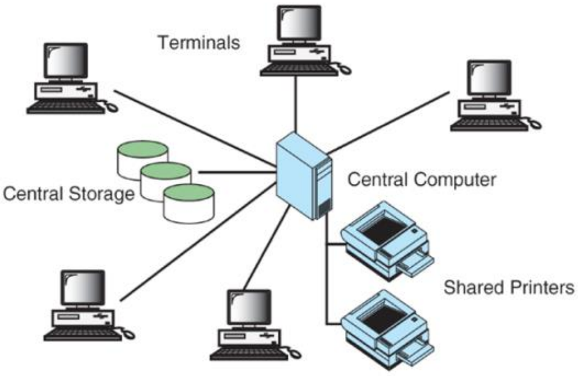


Personal Computing Environment:

In the personal computing environment, there is a single computer system. All the system processes are available on the computer and executed there. The different devices that constitute a personal computing environment are laptops, mobiles, printers, computer systems, scanners etc.

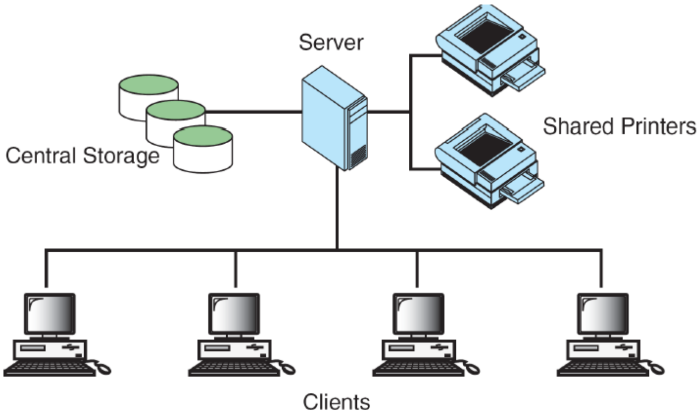
Time Sharing Computing Environment :

The time sharing computing environment allows multiple users to share the system simultaneously. The advantage of time sharing environment is multi programming and multi-tasking at the same time. The users are connected to one or more computers.



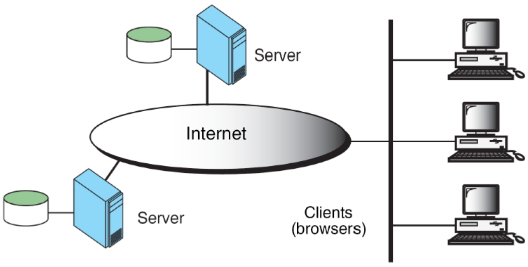
Client-Server Computing Environment :

A client/server system is “a networked computing model that distributes processes between clients and servers, which supply the requested service.” A client/server network connects many computers called clients, to a main computer, called a server. Whenever client requests for something, server receives the request and process it .



Distributed Computing Environment :

A Distributed Computing Environment Provides a seamless integration of computing functions between different servers and clients. the servers are connected by internet all over the world.



Cloud Computing Environment : The computing is moved away from individual computer systems to a cloud of computers in cloud computing environment. The cloud users only see the service being provided and not the internal details of how the service is provided. This is done by pooling all the computer resources and then managing them using a software.

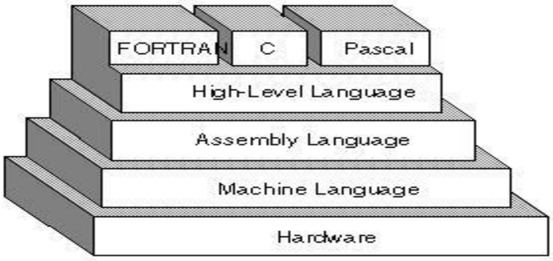
Cluster Computing Environment : The clustered computing environment is similar to parallel computing environment as they both have multiple CPUs. However a major difference is that clustered systems are created by two or more individual computer systems merged together which then work parallel to each other.

1.2 Computer Languages

A programming language: The language used in the communication of computer instructions is known as the programming language. It is a formal constructed language designed to communicate instructions to a machine, particularly a computer.
Programming languages can be used to create programs to control the behavior of a machine.

There are three levels of programming languages are available. They are:

1. Machine languages (Low-level Languages)
2. Assembly languages (Symbolic Languages)
3. High level languages



Machine Languages

A language that is directly understood by the computer without any translation is called machine languages. A machine language is string of 0s and 1s. Machine language are usually referred to as the *first generation* languages. It is also referred to as machine code or object code. Therefore, all instructions and data should be written using *binary codes* i.e., 0's and 1's.Each instruction performs a specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory.

Machine language is the lowest-level programming language. Programs and applications written in low-level language are directly executable on the computing hardware without any interpretation or translation. Machine language and assembly language are popular examples of low level languages.
An instruction written in any machine language has two parts:

i. Operation code (*Opcode*)
ii. Operand address or location

Advantages:

- ✓ Execution speed is very fast
- ✓ Efficient use of primary memory
- ✓ It does not require any translation because machine code is directly understood by the computer.

Disadvantages

- ✓ **Machine dependent:** The machine language is different for different types of computers.
- ✓ **Difficult to write program:** because it requires memorizing dozens of *opcodes* for different commands.
- ✓ **Error prone**: Difficult to modify

Assembly Languages: Assembly languages are usually referred to as the *second-generation* languages. It is also lowest-level programming language. *These* are introduced in 1950's.

- ✓ Assembly languages are just one level higher than machine language.
- ✓ Assembly language consists of special codes called *mnemonics* to represent the language instructions instead of 0's and 1's. The mnemonic code is also called as operation code or *opcode*.
  For example, ADD, SUB, MUL, JMP, LOAD

Advantages

- ✓ **Easier to memorize and use:**Assembly language program is easy to use, understand and memorize because it uses mnemonic codes in place of binary codes
- ✓ **Easy to write input data:** In assembly language programs the input data can be written in decimal number system, later they are converted into binary.
- ✓ It is easier to correct errors and modify program instructions.
- ✓ The Assembly Language has the same efficiency of execution as the machine level language. Because this is one-to-one translator between assembly language program and its corresponding machine language program.
- ✓ Easy print out.
- ✓ Good library facility.

Disadvantages

- ✓ Machine dependent. A program written for one computer might not run in other computers with different hardware configuration.
- ✓ Knowledge of hardware is required.
- ✓ Time consuming
- ✓ Translators required (i.e. **Assembler**)

Assembler: The software used to convert an assembly language programs into machines codes is called an assembler**.**

High Level Languages: Sometimes abbreviated as **HLL**, a **high-level language** is a computer programming language that isn't limited by the computer, designed for a specific job, and is easier to understand. It is more like human language and less like machine language. However, for a computer to understand and run a program created with a high-level language, it must be compiled into machine language.

The first high-level languages were introduced in the 1950's. Today, there are many high-level languages in use, including BASIC, C, C++, COBOL, FORTRAN, Java, Pascal, Perl, PHP, Python, Ruby and Visual Basic.

High-level programming languages can be classified into the following three categories:

1. Procedure-oriented languages (Third generation)
2. Problem-oriented languages (Fourth generation)
3. Natural languages (Fifth generation)

Procedure-oriented languages (Third generation)

A third generation (programming) language (3GL) is a grouping of programming languages that introduced significant enhancements to second generation languages, primarily intended to make the programming language more programmer-friendly.

English words are used to denote variables, programming structures and commands, and Structured Programming is supported by most 3GLs. commonly known 3GLs are FORTRAN, BASIC, Pascal and the C-family (C, C+, C++, C#, Objective-C) of languages.

Also known as 3rd generation language or a high-level programming language.

Problem-oriented languages (Fourth generation)

A fourth generation (programming) language (4GL) is a grouping of programming languages that attempt to get closer than 3GLs to human language, form of thinking and conceptualization.

4GLs are designed to reduce the overall time, effort and cost of software development. The main domains and families of 4GLs are: database queries, report generators, data manipulation, analysis and reporting, screen painters and generators, GUI creators, mathematical optimization, web development and general purpose languages. also known as a 4th generation language, a domain specific language, or a high productivity language.

Natural Languages (Fifth Generation)

A fifth generation (programming) language (5GL) is a grouping of programming languages build on the premise that a problem can be solved, and an application built to solve it, by providing constraints to the program (constraint-based programming), rather than specifying algorithmically how the problem is to be solved (imperative programming).

In essence, the programming language is used to denote the properties, or logic, of a solution, rather than how it is reached. Most constraint-based and logic programming languages are 5GLs. A common misconception about 5GLs pertains to the practice of some 4GL vendors to denote their products as 5GLs, when in essence the products are evolved and enhanced 4GL tools. Also known as 5th generation languages.

Advantages:

- ✓ High-level languages are user-friendly
- ✓ They are easier to learn
- ✓ They are easier to maintain
- ✓ They are problem-oriented rather than 'machine'-based
- ✓ A program written in a high-level language can be translated into many machine languages and can run on any computer for which there exists an appropriate translator
- ✓ The language is independent of the machine on which it is used i.e. programs developed in a high-level language can be run on any computer text
- ✓ They are similar to English and use English vocabulary and well-known symbols
- ✓ A high-level language has to be translated into the machine language by a translator, which takes up time
- ✓ The object code generated by a translator might be inefficient compared to an equivalent assembly language program

1.3 Problem Solving or Program Development Steps

Problem Solving is the process of solving a problem in a computer system by following a sequence of steps.

1. Defining the problem.
2. Designing the algorithm.
3. Coding the program.
4. Testing and debugging the program.
5. Implementing the program.
6. Maintaining and updating the program.

Defining the problem

It involves recognizing the problem; identifying exactly what the problem is; determining the available inputs and desired output; deciding whether problem can be solved by using computer.

Designing the algorithm

An algorithm is finite set of step-by-step instructions that solve a problem. After defining the problem, algorithm can designed.

Coding the program

It involves actually writing the instructions in a particular language that tell the computer how to operate.

Testing and debugging the program

After coding, the program must be tested to ensure that is correct and contains no errors. Three types of errors or bugs can be found which are

1. Syntax error
2. Logical error
3. Runtime error

Syntax Error

Syntax is a set of rules by which a programming language is governed. Syntax error occurs when these rules are violated while coding of the program.

Logical Error

Improper coding of individual statements either or sequence of statements causes this type of computer program. It does not stop the program execution but gives wrong results.

Runtime Error

This error in a computer program stops its execution. It may be caused by an entry of invalid data.

Implementing the Program

After a program has been listed and debugged, it can be installed and implemented.

Maintaining and Upgrading the Program

After a program is developed and implemented can be upgraded as per the requirements of the user. Maintaining and upgrading the program is an ongoing process

1.4 Algorithm:

An algorithm is a finite set of step-by-step instructions to solve a problem. The essential properties of an algorithm are:

1. **Finiteness:** The algorithm must always terminate after a finite number of steps.
2. **Definiteness:** Each and every instruction should be precise and unambiguous i.e. each and every instruction should be clear and should have only one meaning.
3. **Effectiveness:** Each instruction should be performed in finite amount of time.
4. **Input and Output**: An algorithm must take zero or more inputs, and produce one or more outputs.
5. **Generality:** An algorithm applies to different sets of same input type.

Advantages:

1. It is step-by-step solution to a given problem which is very easy to understand.
2. It has got a definite procedure.
3. It is easy to first develop an algorithm, then convert into a flowchart and then into a computer program.
4. It is easy to debug as every step has got its own logical sequence.
5. It is independent of programming languages.

Disadvantages:

1. It is time consuming and cumbersome as an algorithm is developed first which is converted into flowchart and then into a computer program
2. Algorithm lacks visual representation hence understanding the logic becomes difficult.

Examples:

Write an algorithm to add two numbers

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum
Step 3: Read 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
```

Write an algorithm to find the largest among three different numbers

```
Step 1: Start
Step 2: Declare variables A, B and C.
Step 3: Read variables A, B and C.
Step 4: If A>B
            If A>C
                Display A is the largest number.
            else
                Display C is the largest number.
        else
                If B>C
                    Display B is the largest number.
                Else
                    Display C is the largest number.
Step 5: Stop
```

1.5 Flowchart

Flowchart is a pictorial or symbolic representation of an algorithm.It indicates process of solution. They are constructed by using special geometrical symbols. Each symbol represents an activity. Flowcharts are read from top to bottom unless a branch alters the normal flow.

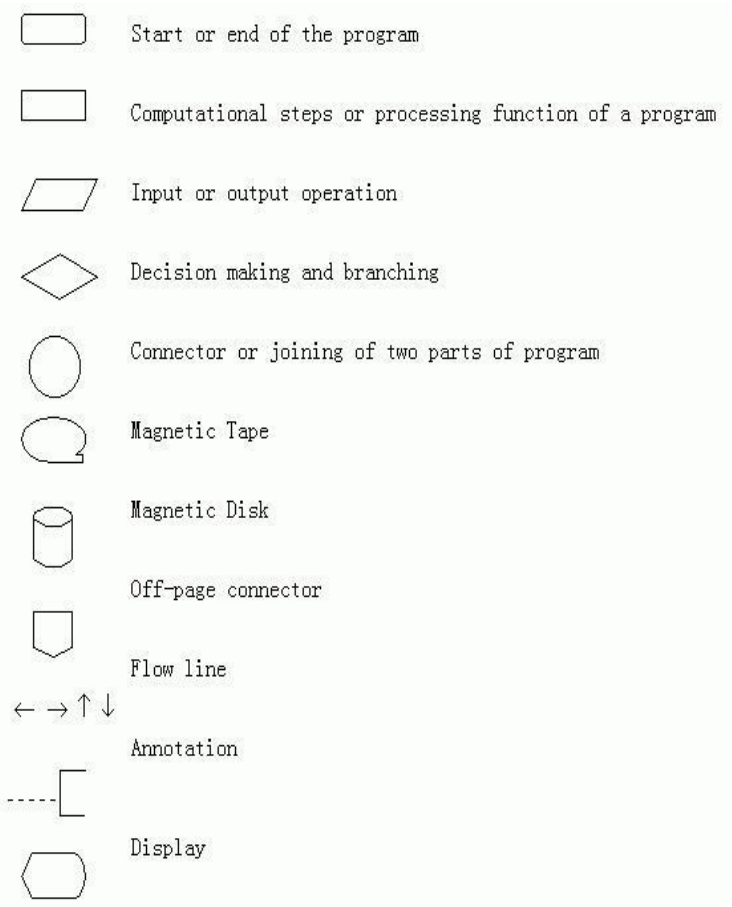
Advantages

1. It clarifies the program logic.
2. Before coding begins the flowchart assists the programmer in determining the type of logic control to be used in a program
3. The flowchart gives pictorial representation
4. Serves as documentation
5. Serves as a guide for program writing.
6. Ensure that all possible conditions are accounted for.
7. Help to detect deficiencies in the problem statement.

Disadvantages:

1. When the program logic is complex the flowchart quickly becomes complex and clumsy and lacks the clarity of decision table.
2. It alterations and modifications are required, the flowchart may require re-drawing completely.
3. As the flowchart symbols cannot be typed reproduction of flowcharts often a problem.
4. It is sometimes difficult for a business person or user to understand the logic depicted in a flowchart.

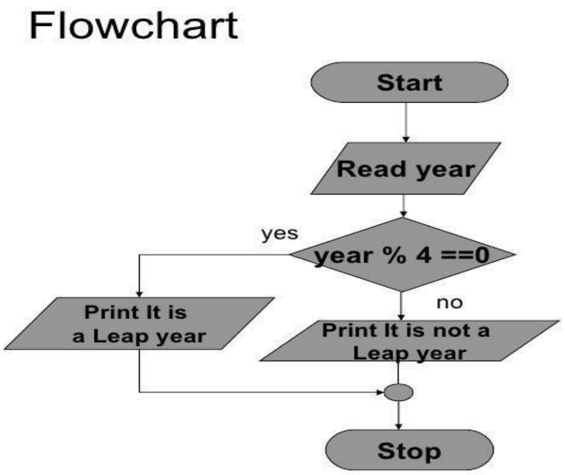
Symbols used to draw flowcharts:


Start or end of the program
Computational steps or processing function of a program
Input or output operation
Decision making and branching
Connector or joining of two parts of program
Magnetic Tape
Magnetic Disk
Off-page connector
Flow line
← → ↑ ↓
Annotation
Display

Comparison between flow chart and algorithm

S.No.	Algorithm	Flowchart
1	An *algorithm* is a finite set of step-by-step instructions that solve problem.	*Flowchart* is a pictorial or symbolic representation of an algorithm.
2.	Algorithm gives verbal which is almost similar to English.	Gives pictorial representation.
3.	Suitable for large and modular programs	Suitable for small programs
4.	Easier to understand	To understand flowcharts one has to familiar with symbols
5.	Drawing tools are not required	Required.
6.	Algorithm can be typed. So reproduction is easy.	Flowchart cannot typed so, reproduction is problem.

Example for a flowchart



Pseudo code: It is a simple way of writing programming code in English. It is a combination of generic syntax and normal English language. Pseudo code is not actual programming language. It uses short phrases to write code for programs before actually create it in a specific language.
It helps the programmer understand the basic logic of the program.

The example of a pseudo code to add two numbers and display the result as shown below:

```
DEFINE:  Integer num1, num2, result
READ:  Integer num1
READ:  Integer num2
SET:  result=num1+num2
Display:  result
```

Advantages

- ✓ Developing program code using *Pseudo code*is easier.
- ✓ The program code instructions are easier to modify in comparison to a flowchart.
- ✓ It is well suited for large program design.

Disadvantages

- ✓ It is difficult to understand the program logic since it does not use pictorial representation.
- ✓ There is no standard format for developing a *Pseudocode*

1.6 Frequently used software components in C:

Editor

In general, an editor refers to any program capable of editing files. The term editor is commonly used to refer to a text editor, which is a software program that allows users to create or manipulate plain text computer files or C source code. For example gedit, notepad, WordPad etc.,

Debugger

This program helps us identify syntax errors in the source code.

Pre Processer

There are certain special instructions within the source code identified by the # symbol that are carried on by a special program called a preprocessor.

Compiler

It is a program which translates a high level language program into a machine language program.

Interpreter

An Interpreter is a program which translates statements of a program into machine code. It translates only one statement of the program at a time.

Linker

The machine code relating to the source code you have written is combined with some other machine code to derive the complete program in an executable file. This is done by a program called the linker.

Loader

Loader is a program that loads machine codes of a program into the system memory. In Computing, a **loader** is the part of an Operating System that is responsible for loading programs.

Difference between Compiler and Interpreter

S.No.	Compiler	Interpreter
1	Compiler takes entire program as a input.	Interpreter takes single instruction as input.
2.	Intermediate object code is generate.	No Intermediate object code is generated.
3.	Conditional control statements are executes faster.	Conditional control statements are executes faster.
4.	Memory required is more. (since object code is generated)	Memory required is less
5.	At a time higher level program is converted into lower level.	Every time higher level program is converted into lower level.
6.	Errors are displayed after entire program is checked. Eg: C compiler	Errors are displayed every instruction interpreted. Eg: BASIC

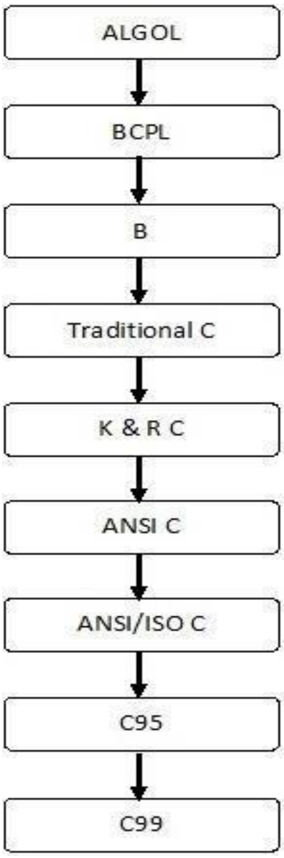
1.7 Introduction to C Language

C language was developed Dennis Ritchie in 1972 at AT&T Bell Laboratories (USA). C language was derived from B language which was developed by Ken Thomson in 1970. This B language was adopted from a language BCPL (Basic Combined Programming Language), which was developed by Martin Richards at Cambridge University. The language B named as so by borrowing the first initial from BCPL language. Dennis Ritchie modified and improved B language and named it as C language, using second initial from BCPL.

C language is not high level language and not a low level language. It is a middle level language with high level and low level features.

C is a general purpose, structured programming language. C language is one of the most popular computer languages today because it is a structured, high level, machine independent language. It allows software developers to develop software without worrying about the hardware platforms where they will be implemented.

The following flowchart shows the history of ANSI C.



Dennis Ritchie

Fig: Taxonomy of C language

Here are most common reasons why C is still so popular:

- ✓ C is one of the most popular programming languages of all time to create system software as well as application software.
- ✓ C is a standardized programming language with international standards.
- ✓ C is the base for almost all popular programming languages.
- ✓ C is one of the foundations for modern computer science and information technology.

Importance of C

It is a robust language whose rich set of built in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a higher level language and therefore it is well suited for both system software and business packages.

Features

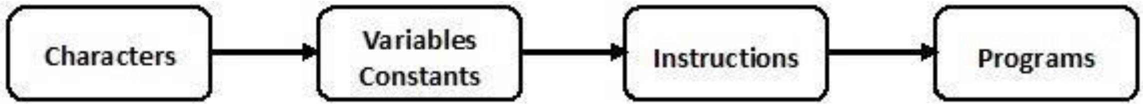
- C is a general purpose language
- C can be used for system programming as well as application programming.
- C is middle level language
- C is portable language
- An application program written on C language will work on many different computers with little or no modifications.
- C Block structured language. In C , a block is marked by two curly braces({ })
- C programs are compact, fast and efficient
- C is case sensitive language
- C is function oriented language
- C language includes advanced data types like pointers, structures, unions, enumerated types etc.
- C language supports recursion and dynamic storage allocation
- C can manipulate with bits, bytes and addresses
- Parameter passing can be done using call-by-value and call-by-reference

Why Use c?

- ***Powerful and flexible***
  C is a powerful and flexible language. The language itself places no constraints on you. C is used for projects as diverse as operating systems, Word Processors, graphics, spreadsheets, and even compilers for other languages.
- ***Popular***
  C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available for use by programmers.
- ***Portable***
  C is a portable language. Portable means that a C program written for one computer system (say an IBM PC) can be compiled and run on another system (say a DEC VAX system) with little or no modification. Portability is enhanced by the ANSI standard for C, the set of rules for C compilers.
- ***Minimum keywords***
  C is a language of few words, containing only a handful of terms, called keywords, which serve as the base on which the language's functionality is built. There is a misconception with many that a language with more key words (sometimes called reserved words) would be more powerful. This isn't true.
- ***Modular***
  C is a modular. C code can (and should) be written in routines called functions. These functions can be reused in other applications or programs. By passing pieces of information to the functions, you can create useful, reusable code.

1.7.1 The C Character Set

We know that English language has 26 alphabets. We will use combinations of these alphabets to form words, sentences, paragraphs etc. In a similar way we will use the available characters while writing programs in any programming language.



There are two set of characters in "C" language

- Source characters
- Execution characters

Source Characters

These are characters that are used to create source text file. Source characters include

Alphabets	A to Z, a to z
Digits	0,1,2,3,4,5,6,7,8,9
Special Characters	, . ; : ? ' " ! | / \ ~ _ $ % # & ^ * + - <> ( ) { } [ ] and blank

Execution Characters

The meaning of these characters is interpreted at the time of execution. These are also known as "Escape sequences because the backslash (\) is considered as an 'escape' character. It causes an escape from normal interpretation of a string. so that the next character is recognized as one having special meaning.

Escape sequences

Escape Sequence	Meaning	Result
\b	Backspace	Moves the cursor to the previous position of the current Line
\a	Bell(alert)	Produces a beep sound for alert
\r	Carriage return	Moves the cursor to beginning of the current line.
\n	New line	Moves the cursor to beginning of the next line.
\0	Null	Null character
\v	Vertical tab	Moves the cursor to next vertical tab position.
\t	Horizontal tab	Moves the cursor to next horizontal tab position.
\\	Backslash	Present a character with backslash(\)

Comments

- ✓ Comments are used by programmers to add remarks and explanations within the program.
- ✓ *Compiler* ignores all the comments and they do not have any effect on the executable program.
- ✓ Comments are of two types; *single line* comments and *multi-line* comments.
- ✓ Single line comments start with two slashes (//) and all the text until the end of the line is considered a comment.

```
// this is a single line comment
```

- ✓ Multi-line comments start with characters /* and end with characters */. Any text between those characters is considered a multi-line comment.

```
/*  this is a multi-line comment and
              Welcome Rgukt              */
```

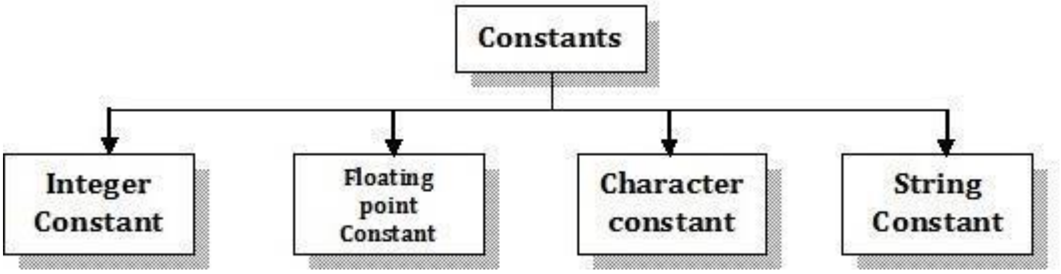
1.8 Various kinds of C Data (Tokens):

The data in C language can be divided into

1. Constants/literals
2. Variables/Identifiers
3. Reserved Words/Key words
4. Delimiters

1. **Constants:**

Constant can be defined as a value that can be stored in memory and cannot be changed during execution of the program. Constants are used to define fixed values like *pi*. C has four basic types of constants. They are:



Integer Constant

An integer constant must have at least one digit and should not have a decimal point. It can be either positive or negative.

Examples for integer constants

1 9 234 999

Floating point Constant

A floating point constant is decimal number that contains either a decimal point or an exponent. In the other words, they can be written in 2 forms: fractional and exponential.

When expressed in fractional form, note the following points.

1. There should be at least one digit, and could be either positive or negative value. A decimal point is must.
2. There should be no commas or blanks.

Examples for fractional form

12.33 -19.56 +123.89 -0.7

When expressed in exponential form, a floating point constant will have 2 parts. One is before *e* and after it. The part which appears before *e* is known as *mantissa* and the one which follows is known as *exponent.* When expressed in this format, note the following points**.**

1. *mantissa* and *exponential* should be separated by letter *E*.
2. *mantissa* can have a positive and negative sign.
3. default is positive.

Examples for fractional form

2E-10 0.5e2 1.2E+3 -5.6E-2

Character Constant

These are single character enclosed in single quotes. A character can be single alphabet, single digit, single special symbol enclosed with in single quotes. Not more than a single character is allowed.

Example

‘a’ ‘Z’ ‘5’ ‘$’

String Constant

A String constant is sequence of characters enclosed in double quotes. So *“a” is* not same as *‘a’.* The characters comprising the string constant are stored in successive memory locations.

Example: “hello” “programming” “cse”

2. Variables/Identifiers

Variable is named memory location that can be used to store values. These variables can take different values but one value at a time. These values can be changed during execution of a program.

Identifiers are basically names given to program elements such as variables, arrays and functions.

Rules for naming a Variable/Identifier

1. The only characters allowed are alphabets, digits and underscore (_).
2. They must start with an alphabet or an underscore (_).
3. It can not include any special characters or punctuation marks (like #, $, ^, ?, ., etc.) except underscore (_).
4. They can be of any reasonable length. They should not contain more than 31 characters. But it is preferable to use 8 characters.
5. There cannot be two successive underscores.
6. Reserved words/keywords cannot be identifiers.
7. Lower case or uppercase letters are significant.

3. Key words:

- ✓ These are also called as reserved words.
- ✓ All Keywords have fixed meanings and these meanings cannot be changed.
- ✓ There are 32 keywords in C programming.
- ✓ Keywords serve as basic building blocks for a program statement.
- ✓ All keywords must be written in lowercase only.

ANSI C Keywords

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*

4. Delimiters

Delimiters are used for syntactic meaning in C. These are given below:

Symbol	Name	Meaning
:	Colon	Used for label
;	Semicolon	End of statement
( )	Parenthesis	Used in expression
[ ]	Square brackets	Used in expression
{ }	Curly braces	Used for block of statements
#	Hash	Pre-processor directive
,	Comma	Variable separator

1.9 Variables Declaration & Initialization in C :

A variable is named memory location that stores a value. When using a variable, we actually refer to address of the memory where the data is stored. C language supports two basic kinds of variables:

1. Numeric variables

2. Character variables

Numeric variables

Numeric variables can be used to store either integer values or floating point values. While an integer value is a whole number without a fraction part or decimal point. A floating point value can have a decimal point.

Numeric values may be associated with modifiers like short, long, signed, and unsigned.

Character variables

Character variables can include any letter from the alphabet or from ASCII chart and numbers 0-9 that are given within single quotes.

Variable declaration

To declare a variable, specify the data type of the variable followed by its name. The data type indicates the kind of data that the variable will store. Variable names should always be meaningful and must reflect the purpose of their usage in the program. In C, variable declaration always ends with a semicolon, for example:

```
char grade;
int emp_no;
float salary;
double bal_amount
unsigned short int acc_no;
```

C allows multiple variable of same type to be declared in one statement, so the following statement is absolutely legal in C

```
float temp_in_deg, temp_in_farh;
```

Initializing variables

Assigning value to variable is called variable initialization. For example:

```
char grade= 'A';
int emp_no=1007;
float salary=8750.25;
double bal_amount=100000000
```

Note: *When variables are declared but not initialized they usually contain garbage values.*

Basically variable declaration can be done in three different places in a C program.

1. Inside main() function are called *local* variables.
2. Outside main() function are called *global* variables.
3. In the function definition header are called *formal* parameters.

Declaring Constants

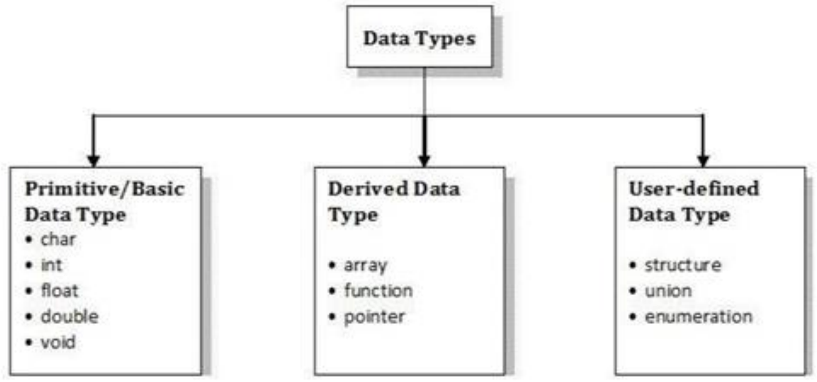
To declare a constant, precede the normal variable declaration with *const* keyword and assign it a value. For example,

```
const float pi=3.1415;
```

This *const* keyword specifies that the value of cannot change.

1.10 Data Types of C :

C supports different types of data. The type or Data type refers a type of data that is going to be stored in variable. Data types can be broadly classified as shown in figure.



C provides a standard minimal set of data types. Sometimes these are also called as 'Primitive types'. They are:

- ✓ '*char*' is used to store any single character.
- ✓ '*int'* is used to store integer value.
- ✓ '*float*' is used to store floating point value
- ✓ '*double*' is used for storing long range of floating point number.

Type Qualifiers

In addition, C has four type qualifiers, also known as type modifiers which precede the basic data type. A type modifier alters the meaning of basic data type to yield a new data type. They are as follows:

- ✓ Short
- ✓ long (to increase the size of an *int* or *double* type)
- ✓ signed
- ✓ unsigned

The qualifiers can be classified into two types:

1. Size qualifier (e.g., *short* and *long*)
2. Sign qualifier (e.g., *signed* and *unsigned*)

Size qualifiers: alters the size of basic data type. The keywords long and short are two size qualifiers. For example:

long int i;

The size of int is 2 bytes but, when long keyword is used, that variable will be either 4 bytes or 8 bytes.

Sign qualifiers: Whether a variable can hold only positive value or both values is specified by sign qualifiers. Keywords *signed* and *unsigned* are used for sign qualifiers.

a. Each of these type modifiers can be applied to base type *int.*
b. The modifiers *signed* and *unsigned* can also be applied to base type *char*.
c. In addition *long* can also be applied to *double.*
d. When base type is omitted from a declaration, *int* is assumed.

Example

Suppose a variable can be assigned only an integer value. Such variable can be declared as

int a;

This declaration reserves two bytes of memory for storing the number. Such a variable can store a number in range -32768 to 32767. In order to increase the range you can add a qualifier to the declaration.

long int a;

The following table shows the basic data types with qualifiers and their ranges.

Data Type	Size (in Bytes)	Range
Char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
Int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
Float	4	3.4E-38 to 3.4E +38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

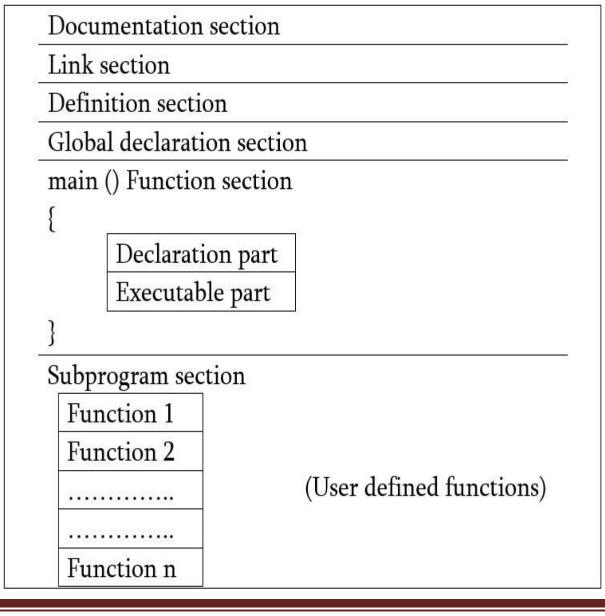
***void* type**

This type holds no value and thus valueless. It is primarily used in three cases:

1. To specify the return type of a function (when the function returns no value)
2. To specify the parameters of the function (when the function accepts no arguments from the caller.
3. To create generic pointers.

1.11 Structure of a C Program

C program is a collection of one or more functions. Every function is collection of statements, performs a specific task. The structure of a basic C program is:



Documentation section

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. There are two types of comment lines.

1. Block comment lines /* */ and
2. Single line comment lines. //

Link section

The link section provides instructions to the compiler to link functions from the system library.

#include directive

The *#include* directive instructs the compiler to include the contents of the file "stdio.h" (standard input output header file) enclosed within angular brackets.

Definition section

The definition section defines all symbolic constants. #define PI 3.1413

Global declaration section

There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.

main() function section

Every C program must have one main function section. This section contains two parts:

1. Declaration part and
2. Executable part

Declaration part: It declares all the variables used in the executable part.

Executable part: There is at least one statement in the executable part.

These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.

Subprogram section

The subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

For Example a ‘C’ program that prints welcome message:

```
#include<stdio.h>
int main( )
{
        printf(“Welcome to C Programming\n”);
        return 0;
}
```

Output

```
Welcome to C Programming.
```

1.12 Formatted input and output functions:

When input and output is required in a specified format the standard library functions *scanf()* and *printf()* are used

Output function printf( )

printf() is a function that is used to print any data on the video monitor screen. It has the following form:

printf("control string",list_of_variables);

The function accepts two arguments - control string, which controls what gets printed, and list of variables.

- ✓ The control string is all-important because it specifies
  - The type of each variable in the list and how user wants it printed. It is also called as format specifier.
  - Characters that are simply printed as they are.
    - Control string that begins with a % sign.
    - Escape sequences that begin with a \ sign
- ✓ List of variables must be separate by comma.
- ✓ The number of format specifiers must exactly match the number of variables.

Format specifiers

Format specifier	Data Type	Display
%c	Char	Single character
%c	unsigned char	
%s		Sequence of characters (String)
%d	Int	Signed integer (both +ve and –ve values)
%u	unsigned int	Unsigned integer (only +ve values)
%hd	short int	Signed short integer
%ld	long int	Long integer
%lu	unsigned long Int	Unsigned long integer (only +ve values)
%o	Int	Octal values
%x	Int	Hexa decimal values
%f	Float	Floating-point value
%lf	Long float/double	Double precision floating-point value
%Lf	long double	Double precision floating-point value
%p	pointer	Address stored in pointer
%%	None	Prints %

Flag characters used in *printf* :

Flag	Meaning
--	Left justify the display
+	Display the positive sign of value
0	Pad with leading zeros
space	Display space if there is no sign

The simplest *printf* statement is

printf("Welcome to the world of C language");

When executed, it prints the string enclosed in double quotes on screen.

Examples:

```
printf("Result:%d%c%f\n",12,'a',2.3);
Result:12a2.3
```

```
printf("Result:%d\t%c\t%f\n",12,'a',2.3);
Result:12        A          2.3
printf("%6d\n",1234);
```

		1	2	3	4

```
printf("%06d\n",1234);
```

0	0	1	2	3	4

```
printf("%-6d\n",1234);
```

1	2	3	4		

```
printf("%-06d\n",1234);
```

-	0	1	2	3	4

```
printf("%2d\n",1234);
```

1	2	3	4

```
printf("%9.2f\n",123.456);
```

			1	2	3	.	4	6

```
char ch='A';
printf("%c\n%2c\n%3c\n",ch,ch,ch);
A
 A
  A
```

```
printf("%X\n",255);
FF
```

Input function *scanf()*

C provides *scanf( )* function for entering input data. This function accepts all types of data as input (numeric, character, string). The general form of *scanf()* is given below

Syntax

scanf("control string",variable1,variable2,…,variable n);

This function should have at least two parameters. First parameter is control string which is conversion specification character. It should be within double quotes. This may be one or more, it depends on the number of variables. The other parameter is variable names.

Each variable must preceded by ampersand (&) sign. This gives starting address of the variable name in memory. *scanf ( )* uses all the format specifiers used in *printf( )* function.

Note: comments are not allowed in scanf()

For example, consider the following simple programs.

```
#include <stdio.h>
int main()
{
        int x;
        printf("Enter number:");
        scanf("%d", &x);
        printf(“The value of x is %d\n”, x);
        return 0;
}
```

Output

```
Enter a number: 45
The value of x is 45
```

For accessing multiple values from keyboard using scanf() function

```
#include <stdio.h>
int main()
{
        int x,y;
        printf(“Enter two numbere:”);
        scanf(“%d%d”, &x,&y);
        printf(“The value of x is %d\n”, x);
        printf(“The value of y is %d\n”, y);
        return 0;
}
```

Output

```
Enter two numbers: 29  47
The value of x is 29
The value of x is 47
```

1.13 Operators in C :

An operator is defined as a symbol that operates on operands and does something. The something may be mathematical, relational or logical operation. C language supports a lot of operators to be used in expressions. These operators can be categorized into the following groups:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Increment/Decrement operators
5. Bitwise operators
6. Conditional operators
7. Assignment operators
8. Special operators

Arithmetic operators

- ✓ These are used to perform mathematical operations.
- ✓ These are binary operators since they operate on two operands at a time.
- ✓ They can be applied to any integers, floating-point number or characters.
- ✓ C supports 5 arithmetic operators. They are +, -, *, /, %.
- ✓ The modulo (%) operator can only be applied to integer operands and cannot be used on float or double values.

Consider three variables declared as,

int a=9,b=3,result;

We will use these variables to explain arithmetic operators. The table shows the arithmetic operators, their syntax, and usage in C language.

Operation	Operator	Syntax	Statement	Result
Addition	+	a+b	result=a+b	12
Subtraction	-	a-b	result=a-b	6
Multiply	*	a*b	result=a*b	27
Divide	/	a/b	result=a/b	3
Modulo	%	a%b	result=a%b	0

a and *b* are called operands.
An Example ‘C’ program that illustrates the usage of arithmetic operators.

```
#include<stdio.h>
int main()
{
	int a=9,b=3;
	printf("%d+%d=%d\n",a,b,a+b);
	printf("%d-%d=%d\n",a,b,a-b);
	printf("%d*%d=%d\n",a,b,a*b);
	printf("%d/%d=%d\n",a,b,a/b);
	printf("%d%%%d=%d\n",a,b,a%b);
	 return 0;
}
```

Output

```
9+3=12
9-3=6
9*3=27
9/3=3
9%3=0
```

Relational operators

A relational operator, also known as a comparison operator, is an operator that compares two operands. The operands can be variables, constants or expressions. Relational operators always return either *true* or *false* depending on whether the conditional relationship between the two operands holds or not.

C has six relational operators. The following table shows these operators along with their meanings

Operator	Meaning	Example
<	Less than	3<5 gives 1
>	Greater than	7<9 gives 0
<=	Less than or equal to	100<=100 gives 1
>=	Greater than or equal to	50>=100 gives 0
==	Equal to	10==9 gives 0
!=	Not equal to	10!=9 gives 1

An example program that illustrates the use of arithmetic operators.

```
#include<stdio.h>
int main()
{
	 int a=9,b=3;
	 printf("%d>%d=%d\n",a,b,a>b);
	 printf("%d>=%d=%d\n",a,b,a>=b);
	 printf("%d<%d=%d\n",a,b,a<b);
	 printf("%d<=%d=%d\n",a,b,a<=b);
	 printf("%d==%d=%d\n",a,b,a==b);
	 printf("%d!=%d=%d\n",a,b,a!=b);
	 return 0;
}
```

Output

```
9 > 3 = 1
9 >= 3 = 1
9 < 3 = 0
9 <= 3 = 0
9= =3 = 0
9 != 3 = 1
```

Logical operators

Operators which are used to combine two or more relational expressions are known as logical operators. C language supports three logical operators – logical AND(&&), logical OR(||), logical NOT(!).

- Logical&& and logical || are binary operators whereas *logical !* is an unary operator.
- All of these operators when applied to expressions yield either integer value 0 (false) or an integer value 1(true).

Logical AND

It is used to simultaneously evaluate two conditions or expressions with relational operators. If expressions on the both sides (left and right side) of logical operators is true then the whole expression is *true* otherwise *false*. The truth table of logical AND operator is given below:

A	**B**	**A&&B**
0	0	0
0	1	0
1	0	0
1	1	1

Logical OR

It is used to simultaneously evaluate two conditions or expressions with relational operators. If one or both the expressions on the left side and right side of logical operators is true then the whole expression is *true* otherwise *false*. The truth table of logical OR operator is given below:

A	**B**	**A||B**
0	0	0
0	1	1
1	0	1
1	1	1

Logical NOT

It takes single expression and negates the value of expression. The truth table of logical NOT operator is given below

A	**!A**
0	1
1	0

For example: x and y are two variables. The following equations explain the use of logical operators.

```
z1 = x&&y     …1
z2 = x||y     …2
z3 =!x        …3
```

Equation1 indicates that z1 is true if both x and y is true. Equation2 indicates z2 is true when x or y or both true. Equation3 indicates z3 is true when x is not true.

An example ‘C’ program that illustrates the use of logical operators

```
        #include<stdio.h>
        int main()
        {
                int z1,z2,z3;
                z1=7>5 && 10>15;
                z2=7>5 || 10>15;
                z3=!(7>5);
                printf(" z1=%d\n z2=%d\n z3=%d\n",z1,z2,z3);
                 return 0;
        }
```

Output

```
 z1=0
 z2=1
 z3=0
```

Unary operators

Unary operators act on single operands. C language supports three unary operators. They are:

1. Unary minus
2. Increment operator
3. Decrement operator

Unary minus

The unary minus operator returns the operand multiplied by –1, effectively changing its sign. When an operand is preceded by a minus sign, the unary minus operator negates its value. For example,

int a, b=10;
a=-(b);

the result of this expression is a=-10.

Increment (++) and decrement (− −) operators

The increment operator is unary operator that increases value of its operand by 1. Similarly, the decrement operator decreases value of its operand by 1. These operators are unique in that they work only on variables not on constants. These are used in loops like while, do-while and for statements.

There are two ways to use increment or decrement operators in expressions. If you put the operator in front of the operand *(prefix)*, it returns the new value of the operand (incremented or decremented). If you put the operator after the operand *(postfix)*, it returns the original value of the operand (before the increment or decrement).

Operator	Name	Value returned	Effect on variable
x++	Post-increment	X	Incremented
++x	Pre-increment	x=x+1	Incremented
x− −	Post-decrement	X	Decremented
− −x	Pre-decrement	x=x – 1	Decremented

An example program that illustrates the use of unary operators

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

```
            printf("x=%d\n", x++);
            printf("x=%d\n",++x);
            printf("x=%d\n",x--);
            printf("x=%d\n",--x);
            printf("x=%d\n",-x);
             return 0;
}
```

Output

```
x=5
x=7
x=7
x=5
x=-5
```

Bitwise operators

As the name suggests, the bitwise operators operate on bits. These operations include:

1. bitwise AND(&)
2. bitwise OR(|)
3. bitwise X-OR(^)
4. bitwise NOT(~)
5. shift left(<<)
6. shift right(>>)

Bitwise AND (&)

When we use the bitwise AND operator, the bit in the first operand is ANDed with the corresponding bit in the second operand. If both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example:

In a C program, the & operator is used as follows.

```
int a=6,b=9,c;
c=a&b;
printf("%d",c);              //prints 0

 0110  (binary equivalent of decimal 6)
 1001  (binary equivalent of decimal 9)
 0000
```

Bitwise OR (|)

When we use the bitwise OR operator, the bit in the first operand is ORed with the corresponding bit in the second operand. If one or both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example:

In a C program, the | operator is used as follows.

```
int a=4,b=2,c;
c=a|b;
printf("%d",c);              //prints 6

  1 0 0           (binary equivalent of decimal 4)
| 0 1 0           (binary equivalent of decimal 2)
  1 1 0
```

Bitwise XOR (^)

When we use the bitwise XOR operator, the bit in the first operand is XORed with the corresponding bit in the second operand. If both bits are different, the corresponding bit in the result is 1 and 0 otherwise. The truth table operator is shown below:

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

For example:

In a C program, the | operator is used as follows.

```
int a=4,b=2,c;
c=a^b;
printf("%d",c);             //prints 6
```

```
 1 0 0          (binary equivalent of decimal 4)
| 0 1 0         (binary equivalent of decimal 2)
--------
 1 1 0
--------
```

Bitwise NOT (~)

One's complement operator (Bitwise NOT) is used to convert each *1* to *0* and *0* to *1*, in the given binary pattern. It is a unary operator i.e. it takes only one operand.

For example, In a C program, the ~ operator is used as follows.

```
int a=4,c;
c=~a;
printf("%d",c);          //prints -5
```

```
~ 1 0 0         (binary equivalent of decimal 4)
--------
   1 0 1
--------
```

Shift operators

C supports two bitwise shift operators. They are shift-left (<<) and shift-right (>>). These operations are simple and are responsible for shifting bits either to left or to the right. The syntax for shift operation can be given as:

operand operator num

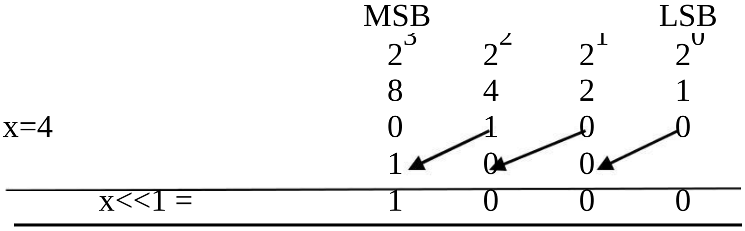
where the bits in *operand* are shifted left or right depending on the *operator* (<<, >>) by the number of places denoted by *num*.

Left shift operator

When we apply left-shift, every bit in *x* is shifted to left by one place. So, the MSB (Most Significant Bit) of x is lost, the LSB (Least Significant Bit) of x is set to 0.

Let us consider int x=4;

Now shifting the bits towards left for 1 time, will give the following result



```
                    MSB                 LSB
                    2^3     2^2     2^1     2^0
                    8       4       2       1
x=4                 0       1       0       0
                    1       0       0
--------------------------------------------------
      x<<1 =        1       0       0       0       the result of x<<1 is 8
--------------------------------------------------
```

Left-shift is equals to multiplication by 2.

Right shift operator

When we apply right-shift, every bit in *x* is shifted to right by one place. So, the LSB (Least Significant Bit) of x is lost, the MSB (Most Significant Bit) of x is set to 0. Let us consider *int x=4;*
Now shifting the bits towards right for 1 time, will give the following result.

	2^3	2^2	2^1	2^0	
	8	4	2	1	
x=4	0	1	0	0	
		0	1	0	
x>>1 =	0	0	1	0	the result of x>>1 is 2

Right-shift is equals to division by 2.

An example 'C' program that illustrates the use of bitwise operators

```
#include <stdio.h>
int main()
{
        int a=4,b=2;
        printf("%d | %d = %d\n",a,b,a|b);
        printf("%d & %d = %d\n",a,b,a&b);
        printf("%d ^ %d = %d\n",a,b,a^b);
        printf("~%d = %d\n",a,~a);
        printf("%d<<1 = %d\n",a,a<<1);
        printf("%d>>1 = %d\n",a,a>>1);
        return 0;
 }
```

Output:

```
4 | 2 = 6
4 & 2 = 0
4 ^ 2 = 6
~4 = -5
4 << 1 = 8
4 >> 1 = 2
```

Assignment operators

Assignment operator (=)

In C, the assignment operator (=) is responsible for assigning values to variables. When equal sign is encountered in in an expression, the compiler processes the statement on the right side of the sign and assigns the result to variable on the left side. For example,

```
int x;
x=10;
```

Assigns the value 10 to variable x. if we have,

```
int x=2,y=3,sum=0;
sum = x + y;
then sum=5.
```

The assignment has right-to-left associativity, so the expression

int a=b=c=10;

is evaluated as

(a=(b=(c=10)))

Consider the following set of examples:

```
  a = 5;                    // value of variable 'a' becomes 5
```

```
a = 5+10;            // value of variable ‘a’ becomes 15
a = 5 + b;           // value of variable ‘a’ becomes 5 + value of b
a = b + c;           // value of variable ‘a’ becomes value of b + value of c
```

Short hand/Compound assignment operators

The assignment operator can be combined with the major arithmetic operators such operators are called compound assignment operators. C language supports a set of compound assignment operators of the form:

variable op = expression;

where *op* is binary arithmetic operator.

The following table lists the assignment operators supported by the C:

Op	Description	Example

+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to
		C = C + A

−=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C −= A is equivalent to C = C − A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

An example ‘C’ program that illustrates the use of assignment operators

```
#include <stdio.h>
int main()
{
     int a = 21,c;
     c = a;
     printf("Operator is = and c = %d\n", c ); c += a;
     printf("Operator is += and c=%d\n",c); c −= a;
     printf("Operator is −= and c=%d\n",c); c *= a;
     printf("Operator is *= and c=%d\n",c); c /= a;
     printf("Operator is /= and c=%d\n", c);
     iii. = 200; c %= a;
     printf("Operator is %= and c=%d\n",c);    c <<= 2;
   printf("Operator is <<= and  c=%d\n",c);
   c >>= 2;
   printf("Operator is >>= and  c=%d\n",c);
  c &=  2;
```

```
    printf("Operator is &= and  c = %d\n", c );
    c ^= 2;
    printf("Operator is ^= and  c = %d\n", c );
    c |= 2;
    printf("Operator is |= and  c = %d\n", c );
    return 0;
}
```

Output

```
Operator is = and c = 21
Operator is += and  c = 42
Operator is −= and  c = 21
Operator is *= and c = 441
Operator is /= and c = 21
Operator is %= and  c = 11
Operator is <<= and c = 44
Operator is >>= and c = 11
Operator is &= and c = 2
Operator is ^= and c = 0
Operator is |= and c = 2
```

Advantages:

1. Short hand expressions are easier to write.
2. The statement involving short hand operators are easier to read as they are more concise.
3. The statement involving short hand operators are more efficient and easy to understand.

Conditional operator (? :)

It is also called *ternary* operator because it takes three operands. It has the general form:

variable = expression1 ? expression2 : expression3;

If the *expression1* is evaluated to *true* then it evaluates *expression2* and its value is assigned to variable, otherwise it evaluates *expression3* and its value is assigned to variable.

It is an alternative to simple *if..else* statement

For example

```
#include<stdio.h>
int main()
{
        int a=5,b=6,big;
        big = a>b ? a : b;
        printf("%d is big\n", big);
        return 0;
}
```

Output

```
        6 is big
```

Special operators: Comma operator (,)

This operator allows the evaluation of multiple expressions, separated by the *comma*, from left to right in the order and the evaluated value of the rightmost expression is accepted as the final result. The general form of an expression using a comma operator is

expressionM= (expression1, expression2, expressionN);

For example

```
#include<stdio.h>
int main()
{
        int k=5,i,j;
        i=k++, j=(k++,k++,++k);
        printf(“i=%d\t j=%d”, i,j);
        return 0;
}
```

Output: i=5 j=9

```
#include<stdio.h>
 int main()
 {
        int k=5,i,j;
        i=k++, j =k++, k++, ++k;
        Printf(“i=%d \t j=%d”, i,j);
        return 0;
```

Output:i=5   j=6

Sizeof() Operator

The operator *sizeof* is a unary operator used calculates the size of data types and variables. The operator returns the size of the variable, data type in bytes. i.e. the *sizeof* operator is used to determine the amount of memory space that the variable/data type will take. The outcome is totally machine-dependent. For example:

```
#include<stdio.h>
int main()
{
        printf("char occupies %d bytes\n",sizeof(char));
        printf("int occupies %d bytes\n",sizeof(int));
         printf("float occupies %d bytes\n",sizeof(float));
         printf("double occupies %d bytes\n",sizeof(double));
        printf("long double occupies %d bytes\n",sizeof(long double));
        return 0;

  }
```

Output

```
char occupies 1 bytes
int occupies 2 bytes
float occupies 4 bytes
double occupies 8 bytes
long double occupies 10 byte
```

1.14 Expressions :

In C programming, an expression is any legal combination of operators and operands that evaluated to produce a value.Every expression consists of at least one *operand* and can have one or more *operators*. Operands are either variables or values, whereas operators are symbols that represent particular actions.

In the expression x + 5; x and 5 are operands, and + is an operator.

In C programming, there are mainly two types of expressions are available. They are as follows:

1. Simple expression
2. Complex expression

Simple expression: In which contains one operator and two operands or constants.
Example: x+y; 3+5; a*b; x-y etc.

Complex expression: In which contains two or more operators and operands or constants.
Example: x+y-z; a+b-c*d; 2+5-3*4; x=6-4+5*2 etc.

Operators provided mainly two types of properties. They are as follows:

1. Precedence and
2. Associativity

1. Operator Precedence

It defines the order in which operators in an expression are evaluated depends on their relative precedence. Example: Let us see x=2+2*2

1^{st} pass- 2+**2*2**
2^{nd} pass- **2+4**
3^{rd} pass- 6 that is x=6.

2. **Associativity**

It defines the order in which operators with the same order of precedence are evaluated. Let us see x=2/2*2

1^{st} pass-- **2/2***2
2^{nd} pass-- **1*2**
3^{rd} pass-- **2**that is x=2

The below table lists C operators in order of *precedence* (highest to lowest) and their *associativity* indicates in what order operators of equal precedence in an expression are applied.

S.No	Operators	Associativity
1	**( ) [ ] → . ++ (postfix) − −(postfix)**	L to R
2	**++ (prefix) -- (prefix) ! ~ sizeof(type) + (unary) − (unary) &(address) * (indirection)**	R to L
3	*** / %**	L to R
4	+ −	L to R
5	<<>>	L to R
6	<<= >>=	L to R
7	= = **!=**	L to R
8	**&**	L to R
9	^	L to R
10	|	L to R
11	**&&**	L to R
12	||	L to R
13	**? :**	R to L
14	= += −= *= /= **%=** >>= <<= **&=** ^= |=	R to L
15	**,**	L to R

An example ‘C’ program that illustrates the use of expressions

```
 #include<stdio.h>
 int main()
 {
          int a, b=4,c=8,d=2,e=4,f=2;
          a=b+c/d+e*f;                              // expression1
          printf(" The value of a is%d\n", a);
          a=(b+c)/d+e*f;                            // expression2
          printf("The value of a is%d\n", a);
          a=b+c/((d+e)*f);       //expression 3
          printf("The value of a is%d\n", a);
          return 0;
}
```

Output:

```
The value of a is 16
The value of a is 14
The value of a is 6
```

1.15 Type Conversion:

Type casting: Type casting is a way to convert a variable from one data type to another data type. It can be of two types:

1. Implicit type casting
2. Explicit type casting

Implicit type casting

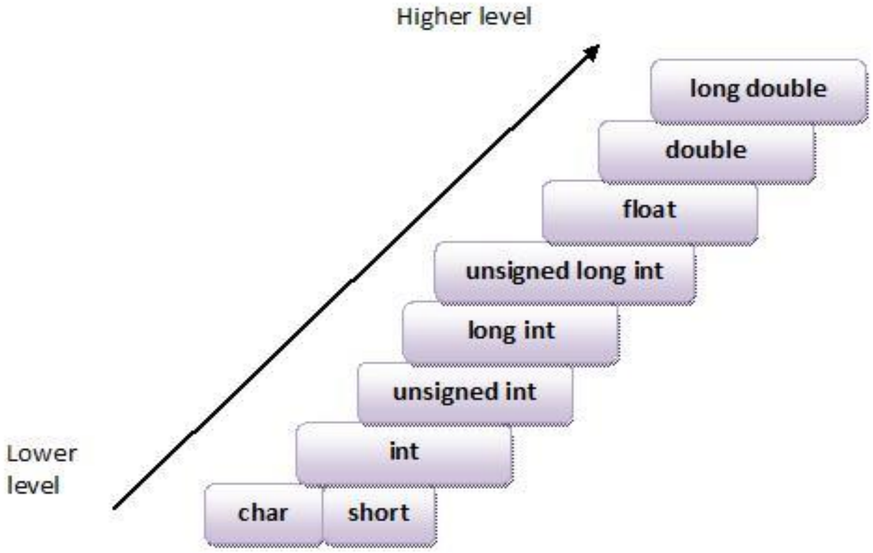
When the type conversion is performed automatically by the compiler without programmer’s intervention, such type of conversion is known as **implicit type conversion** or **type promotion**. In this, all the lower data types are converted to its next higher data type.

If the both operands are of the same type, promotion is not needed. If they are not, promotion follows these rules:

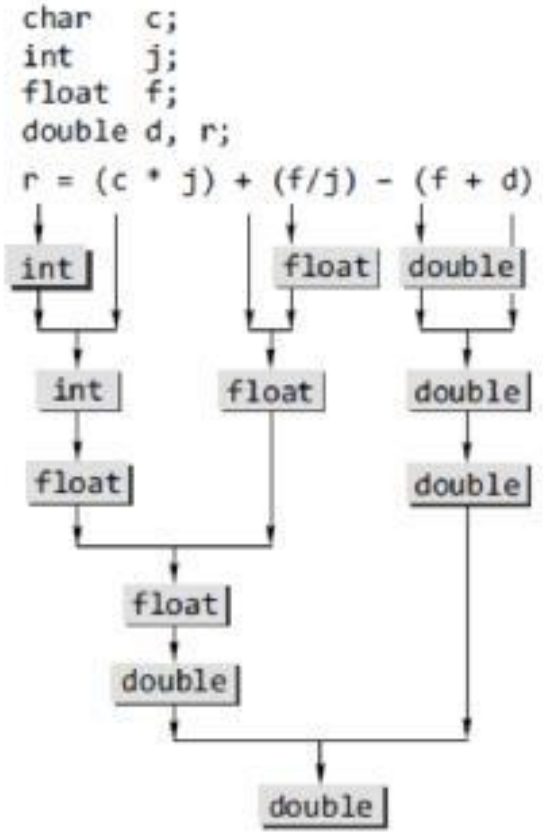
- ✓ *float* operands are converted to *double*.
- ✓ *char or short* are converted to *int*.
- ✓ If anyone operand is *double*, the other operand is also converted to *double*, and that is the type of result.
- ✓ If anyone operand is *long*, the other operand is also converted to *long*, and that is the type of result.
- ✓ If anyone operand is *unsigned*, the other operand is also converted to *unsigned*, and that is the type of result

.

The smallest to the largest data types with respect to size are given as follows:



Conversion hierarchy of data types



```
char   c;
int    j;
float  f;
double d, r;
r = (c * j) + (f/j) - (f + d)
```

Fig. Conversion of types in a mixed expression

A Sample 'C' program that illustares the use implicit type conversion

```
#include<stdio.h>
int main()
{
	int sum, num=17;
	char ch='A';
	sum=num+ch;
	printf("The value of sum=%d\n", sum);
	return 0;
}
```

Output:

The value of sum= 82 i.e. sum=num+ch=>17+65 (ASCII value of 'A')

Explicit typecasting: Which is intentionally performed by the programmer for his requirement in a C program? The explicit type conversion is also known as **type casting**.We can convert the values from one type to another explicitly using the **cast operator** as follows:

Syntax

(data_type) expression;

Where, *data_type* is any valid C data type, and *expression* may be constant, variable or expression.

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

1. All integer types to be converted to float.
2. All float types to be converted to double.
3. All character types to be converted to integer.

Let us look at some examples of type casting.

res = (int) 9.5;
9.5 is converted to 9 by truncation then assigned to *res.*

res = (int) 12.3 / (int) 4.2 ;
It evaluated as 12/4 and the value 3 is assigned to *res.*

res = (double) total / n;
total is converted to *double* and then division is done in float point mode.

res = (int)(a + b);
The value of *(a + b)* is converted to *integer* and then assigned to *res.*

res = (int)a + b;
a is converted to *int* and then added with *b.*

1.16 Mathematical Functions in C

The mathematical functions such as *sin, cos, sqrt, log* etc., are frequently used in analysis of real-life problems. Most of the C compilers support these basic math functions. The following Table lists some standard math functions

Function	Meaning
Trigonometric	
asin(x)	Arc sin of x.
acos(x)	Arc cosine of x
atan(x)	Arc tangent of x
atan2(y,x)	Arc tangent of y/x
sin(x)	sine of *x*

cos(x)	cosine of *x*
tan(x)	tangent of *x*
Hyperbolic	
sinh(x)	hyperbolic sine of *x*
cosh(x)	hyperbolic cosine of *x*
tanh(x)	hyperbolic tangent of *x*
Other functions	
exp(x)	*e* to the power of *x* (e^x)
ceil(x)	*x* rounded up to the nearest integer.
floor(x)	*x* rounded down to the nearest integer
fabs(x)	absolute value |x|
log(x)	Natural logarithm of x, *x*>0.
log10(x)	Base 10 logarithm x, *x*>0.
fmod(x,y)	Remainder of x/y
sqrt(x)	square root of x, x>=0.
pow(x,y)	*x* to the power *y*.

Note:

1. ***x*** and ***y*** should be declared as ***double***.
2. In trigonometric and hyperbolic functions, ***x*** and ***y*** are ***radians***.
3. All the functions return a ***double.***

We should include the line:

#include<math.h>

in the beginning of the program

An example ‘C’ program that illustrates the use mathematical functions

```
#include <stdio.h>
#include <math.h>
int main()
{
   printf("sin 90 = %.0f\n",sin(90));
   printf("cos 0= %.0f\n",cos(0));
   printf("sqrt(9) = %.0f\n",sqrt(9));
   printf("floor(9.56) = %.2f\n",floor(9.56));
    printf("ceil(9.56) = %.2f\n",ceil(9.56));
   printf("abs(-9) = %.2f\n",fabs(-9));
   printf("power(2,3) = %.0f\n",pow(2,3));
   printf("Remainder of 9/3 = %.0f\n",fmod(9,3));
   return 0;

}
```

Output

```
sin 90 = 1
cos 0= 1
sqrt(9) = 3
floor(9.56) = 9.00
ceil(9.56) = 10.00
abs(-9) = 9.00
power(2,3) = 8
Remainder of 9/3 = 0
```