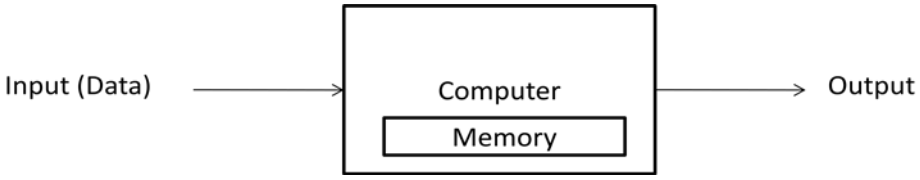
# UNIT - 1

## COMPUTER BASICS

## INTRODUCTION TO C

## Computer

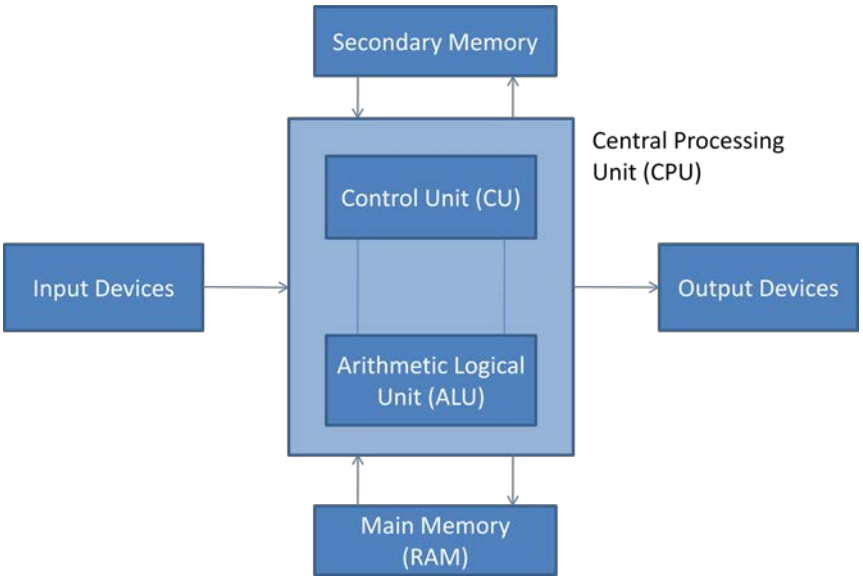
A computer is an electronic device which accepts data as input, performs operations on the data based on the instructions stored in the memory and produces output. General representation of a computer is as shown below:




A computer is made up of two components: 1) Hardware and 2) software. The computer hardware is the physical equipment which is made of electronic components like registers, transistors, capacitors etc.. The software is the collection of programs that allow the hardware to serve its purpose. The purpose of the software is to use the underlying hardware components.

## Architecture of a Computer

The architecture or structure of a computer is as shown below:




The essential hardware components of a computer are:
1) CPU (Central Processing Unit)
2) Memory
3) I/O (Input/Output) Devices

**Central Processing Unit (CPU):** The CPU is the core hardware component of a computer. It is responsible for executing instructions such as arithmetic calculations, comparisons among data and movement of data inside the system.

The Central Processing Unit mainly consists of two components: 1) Arithmetic Logical Unit (ALU) and 2) Control Unit (CU).

***Arithmetic Logical Unit:*** The ALU performs arithmetic and logical operations on the data based on the instructions stored in the memory.

***Control Unit:*** The Control Unit (CU) coordinates the system components, like transfer of data between the components, timing etc.

**Memory:** Memory is the place where the programs and data are stored temporarily during processing. There are two types of memory in the computer system. They are: 1) Main memory or Primary memory and 2) Auxiliary memory or Secondary memory.

***Main memory:*** This is the place where the programs and data are stored temporarily during processing. The data in the main memory are erased when we turn off the computer or when we log off.

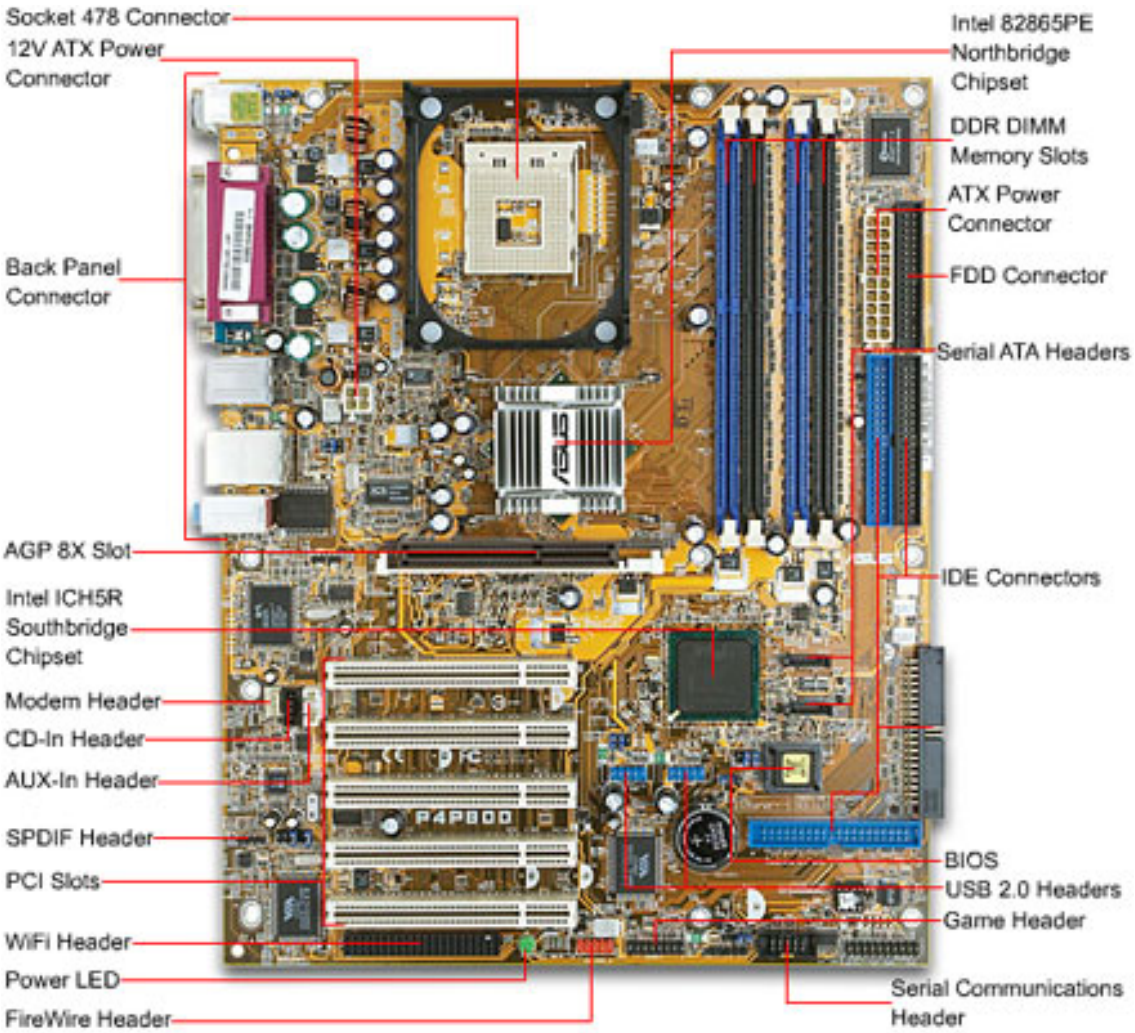
***Auxiliary memory:*** This memory is the place where the programs and data are stored permanently. When we turn off the computer, our programs and data remain in the secondary storage, ready for the next time we need them.

**I/O Devices:** The input devices allow the user to give data as input to the computer and the output devices allow the computer to show information to the user. Examples of input devices are: keyboard, mouse, and scanner. Examples of output devices are monitor, speakers and printer.

# Hardware Components inside a Computer

## Motherboard

Motherboard is the most important hardware component of the computer. Motherboard is the hardware component that holds the other hardware components together as a single unit. The motherboard, or mainboard, of a PC is a large circuit board that is home to many of the most essential parts of the computer like: microprocessor, chipset, cache, memory sockets, bus, parallel and serial ports, mouse and keyboard connectors, hard disk and floppy disk sockets etc.



## Microprocessor

Microprocessor or Processor or Central Processing Unit is the heart of every computer. It is designed to perform all of the arithmetic, logic and other basic computing steps that make up the actions of your computer. Whenever a user executes a word processor, a computer game, a browser or any other software program, the CPU performs hundreds or even thousands of instructions. The microprocessor is a piece of electronic circuitry that uses digital logic to perform the instructions of the software.


A microprocessor is an integrated circuit – a thin slice of silicon crystal packed with microscopic circuit elements like: wires, transistors, capacitors, resistors.

## Random Access Memory (RAM)

The Random Access Memory (RAM) is used to store data, instructions (programs) which are accessed by the CPU for execution. RAM is also known as main memory or primary memory. The data or instructions stored on the RAM can be accessed at random. Hence the name Random Access. RAM is known as volatile memory means, the data or information stored on the RAM is wiped out once the computer is turned off or the power goes off.

RAM is of two types namely, Static RAM (SRAM) and Dynamic RAM (DRAM). In SRAM the electronic components used are flip-flops. The size of flip-flop is generally bigger than other electronic components like capacitors and transistors. Hence the size of SRAM's is smaller when compared to DRAM's. Also SRAM is faster when compared to DRAM. DRAM is made of capacitors and transistors. DRAM requires to be refreshed at regular intervals to retain the data. Hence DRAM is slower when compared to SRAM. DRAM's are available in larger sizes when compared to SRAM's. The widely used type of RAM being used now-a-days is DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory). There are various in this type of RAM like: DDR2 SDRAM and DDR3 SDRAM.

## Read Only Memory (ROM)

Read Only Memory is a non-volatile memory in the computer. It is used to store firmware (software which is tightly coupled with the hardware). As the name implies this memory can only be read and the data stored on it is permanent i.e., data is retained even when the computer is turned off or the power goes off. Generally ROM's are used to store BIOS (Basic Input Output System) also known as firmware.

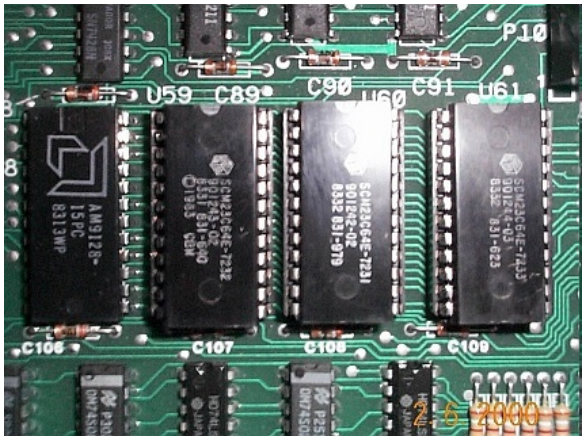
Read Only Memory is of two types: masked ROM's and non-masked ROM's. Masked ROM's are programmed before they are available in the market. Once they are purchased, they cannot be reprogrammed. Non-Masked ROM's can be reprogrammed even after they are purchased from the market. There are three types of non-masked ROM's. They are:

1) PROM (Programmable Read Only Memory)
2) EPROM (Erasable Programmable Read Only Memory)
3) EEPROM (Electronically Erasable Programmable Read Only Memory)

**PROM:** Programmable Read Only Memory (PROM) or also called as One Time Programmable (OTP) ROM can be written to via special device known as PROM programmer. PROM can be programmed only once.

**EPROM:** This type of ROMs can be erased by exposing them to strong Ultra Violet rays. These ROMs can be programmed multiple times. Continuous exposure to UV rays will result in wearing out the ROM and making it unusable.

**EEPROM:** This type of ROM allows the entire ROM or selected banks of memory to be erased and rewritten electrically. There is no need for EEPROMs to be removed from the motherboard to be reprogrammed. EEPROMs can be reprogrammed multiple times.




The ROM must store the data permanently. The data must not be wiped out when the computer is turned off or the power goes off. To help the ROM to store its data permanently, motherboard consists of a CMOS battery, which supplies power continuously to the ROM.

## Secondary Memory or Auxiliary Memory

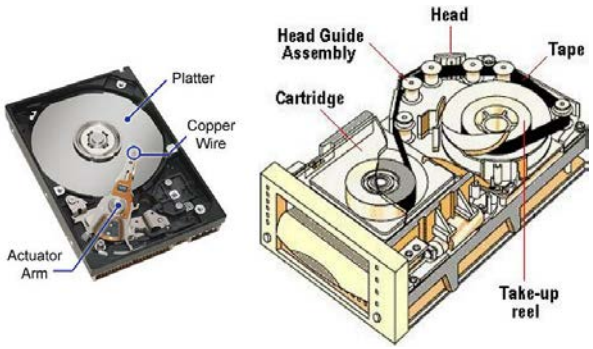
Secondary memory is not directly accessible by the CPU. The secondary memory of a computer includes: Hard disk, Floppy disk, CD, DVD, Blue ray disc, Pen drives, SD cards, tape devices and other types of USB memory devices. The secondary memory is used to store data permanently. The secondary memory is non-volatile memory.

## Virtual Memory

When the primary memory (RAM) is filled with data and instructions, and a new process has no memory to be allocated in the RAM, then the CPU shifts the least recently used blocks of memory onto secondary memory and allocates space to the new process in the RAM. Such memory in the secondary memory which is used as if it is primary memory is known as virtual memory. When virtual memory is used by the CPU, it moves data into and out of virtual memory which is generally slower than accessing data directly from the RAM. Hence, when virtual memory is used, system performance is degraded.

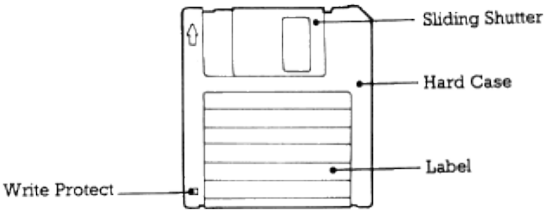
## Magnetic Tape Devices

Magnetic tape is a medium for magnetic recording, made of a thin magnetizable coating on a long thin strip of plastic film. Devices that record and playback audio and video using magnetic tape are called as audio tape recorders and video tape recorders. A computer device that stores data on magnetic tape is called as tape drive. The data stored on a tape drive has to be accessed sequentially.



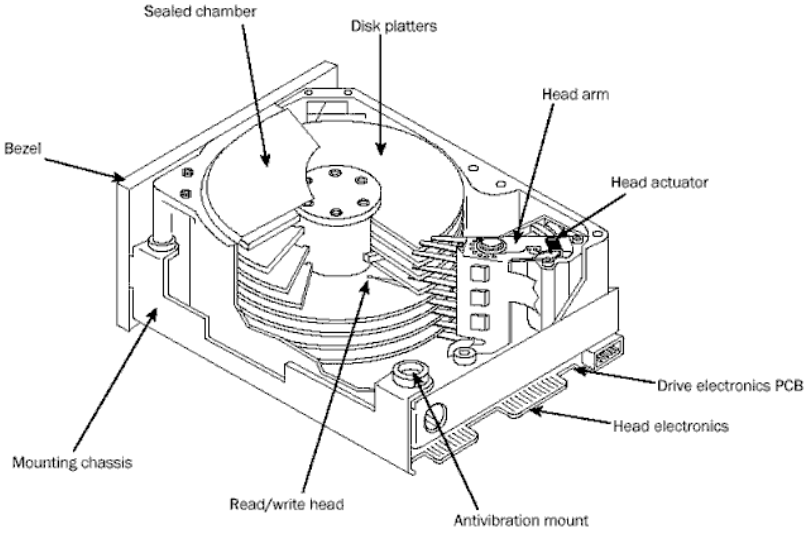

## Floppy Disk

Floppy disk or diskette is an example for secondary memory. It is available in 3.5inch and 5.25inch sizes commonly. In olden days, the floppy disk was the primary data storage device o the PC, but it was later moved to a role of removable media for single files or small collections of files. The floppy disk still has a role for transferring data from one PC to another, backing up small files and compressed file and device driver distribution, although this is also moving to CD-ROM or downloading.




## Hard Disk

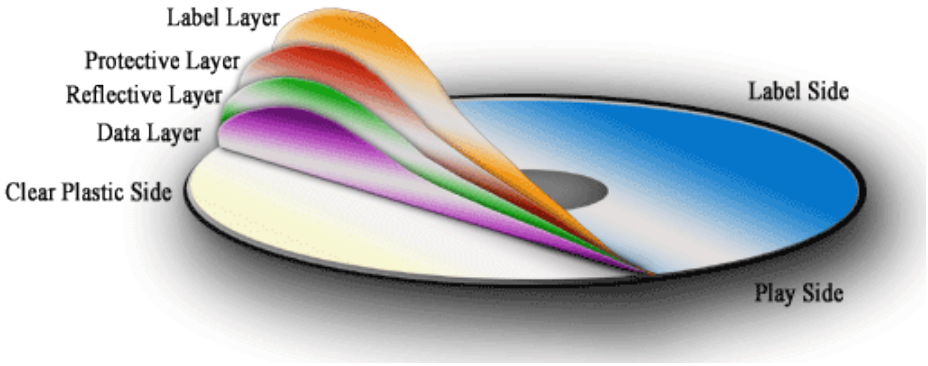
Hard Disk is a device for storing and retrieving digital information, when needed, primarily computer data. It consists of hard rapidly rotating discs coated with magnetic material, and with magnetic heads arranged to write data on to the surfaces and read it from there later when needed. Hard disk is non-volatile memory where the data can be stored permanently.



## CD/DVD Disc

The Compact Disc or in short CD is an optical disc which is used to store digital data. It was originally developed to store and play back sound recordings only, but the format was later adapted for storage of data (CD-ROM), write-once audio and data storage (CD-R), rewritable media (CD-RW), Video Compact Discs (VCD), Super Video Compact Discs (SVCD), PhotoCD, PictureCD, CD-i, and Enhanced CD. The standard capacity of a CD is 700MB.

Digital Versatile Disc or in short DVD is also an optical storage medium like CD but DVDs have larger storage capacities than CD. Both CDs and DVDs are read and written by using laser technology.




## Cache Memory

In a computer, generally the CPU accesses the data from Random Access Memory (RAM). There is another faster memory than RAM in the computer where small amount of information can be stored. Such memory is known as cache memory. Cache memory is smaller and costlier than RAM. The most frequently accessed data or instructions are stored in cache memory. CPU first checks for data in cache, and if not found then searches the RAM for data. Cache memory is generally available on the motherboard.

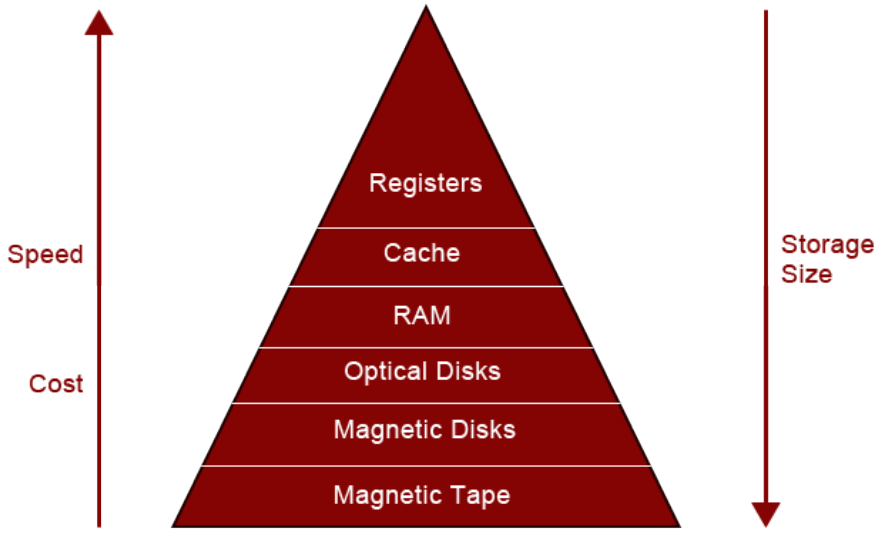
## Magnetic Vs Optical Discs

Examples for magnetic devices are magnetic tape discs, hard discs and floppy disc. Examples for optical discs are CDs, DVDs and Blue ray discs. Let's see the difference between magnetic and optical technologies:

| Magnetic | Optical |
|---|---|
| 1. Stores data in magnetic form. | 1. Stores data as dents by using a laser. |
| 2. Data is affected by magnetic field. | 2. Data is not affected by magnetic field. |
| 3. High storage capacity. | 3. Less storage capacity. |
| 4. Does not use laser to read and write. | 4. Uses a laser to read and write. |
| 5. Easy to modify data. | 5. Difficult to modify data. |

## Memory Hierarchy

In the computer there are various types of memory for storing digital data like: RAM, Hard disk, CD, DVD, Blue ray, Flash Storage, Cache and Registers etc. Let's see the memory hierarchy which specifies which memory is slower; which type of memory is cheaper and which memory allows us to store maximum amount of data.




As seen in the above figure, the storage capacity increases from top to bottom. The speed and cost increases from bottom to top. So the devices which can hold maximum amount of data are magnetic disks and magnetic tape discs. The fastest memory is registers and the costliest memory is cache and registers.

## BIOS (Basic Input Output System)

BIOS is software which is used for starting up the computer. The importance of BIOS is that it performs all of the functions the PC needs to get started. BIOS contains the first instruction the computer needs to get started, programming that checks that computer's hardware is attached and ready, and other routines to help the computer get up and running. A computer's BIOS include instructions to perform three vital and useful functions for the PC:
1) It boots the computer.
2) It validates computer's configuration.
3) It provides an interface between the hardware of the PC and its software.

## Booting

The procedure (set of operations) performed right from powering on the computer till the computer is ready to be used is called as booting. The boot process is performed under the guidance of the BIOS. The BIOS contains the instructions needed to verify, test and start the computer. The computer's hardware cannot perform operations on its own. It must have instructions to do anything at all which is provided by the software. The boot process involves the following:

- Power initialization
- BIOS startup
- POST signal (Power-On Self-Test)
- Video and Device BIOS check
- System check
- Plug-and-Play check
- Boot device found
- Operating System is loaded and running

When the system is booted when the computer is in power off state, then it is known as *cold boot*. If the system is booted when the computer is already running, then it is known as *warm boot*.

## Chipset

The chipset resides on the motherboard which provides communication between the various components available on the motherboard. Without chipset, the computer cannot be operated.

### SMPS

Switched-Mode Power Supply (SMPS) is the hardware component which supplies power to other components in the computer. Initially the power from the main reaches the SMPS and then power reaches the motherboard from SMPS which supplies power to all other hardware components.

## Advantages of Computers

Computer is a powerful electronic device which can be used for performing many operations. Let us see why computer is very powerful or the advantages of using computers.

1) Speed
2) Accuracy
3) Reliability
4) Storage
5) Communication

## Applications of Computers

Computers can be used for many purposes. Now-a-days computers can be seen almost anywhere. Some of the areas in which computers can be used are:

1) Artificial Intelligence
2) Virtual Reality
3) Augmented Reality
4) Animation
5) Game Design
6) CAD/CAM
7) Information Processing
8) Robotics

## Software Concepts

A computer generally consists of two components: Hardware and Software. The purpose of software is to use the hardware components. So far, we have seen the hardware components. Now, let's learn about the software related concepts.

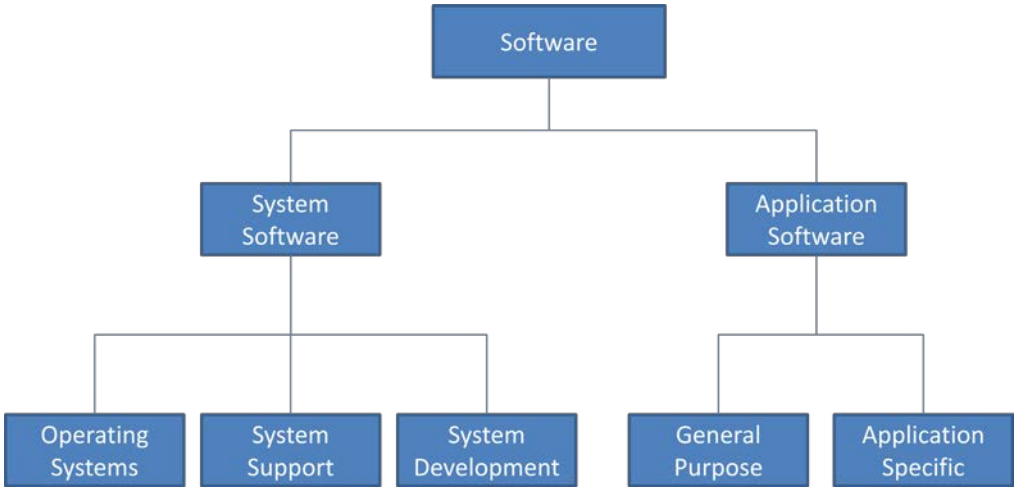
### Program

A program is a set of instructions for solving a particular problem. Using programs a human can use the underlying hardware components like CPU, Memory etc.

### Software

Software is a set of programs. Computer software is divided into two broad categories: 1) System software and 2) Application software. System software manages the computer's hardware resources. It provides the interface between the hardware and the users but does nothing for users to use the hardware. Application software, on the other hand, is directly responsible for helping users solve their problems.

The various types of software are as shown in the below diagram:




**System Software:** This consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: 1) Operating System, 2) System Support and 3) System Development.

**1) Operating System:** Operating system provides the interface between the application software/user and hardware. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system. Examples: Windows XP, Vista, Seven, Linux, UNIX, MAC OSX, MAC Lion etc.

**2) System Support:** This provides utilities and other operating services to work with hardware. This software comes along with the operating system. Examples: Disk format software, Sort programs etc.

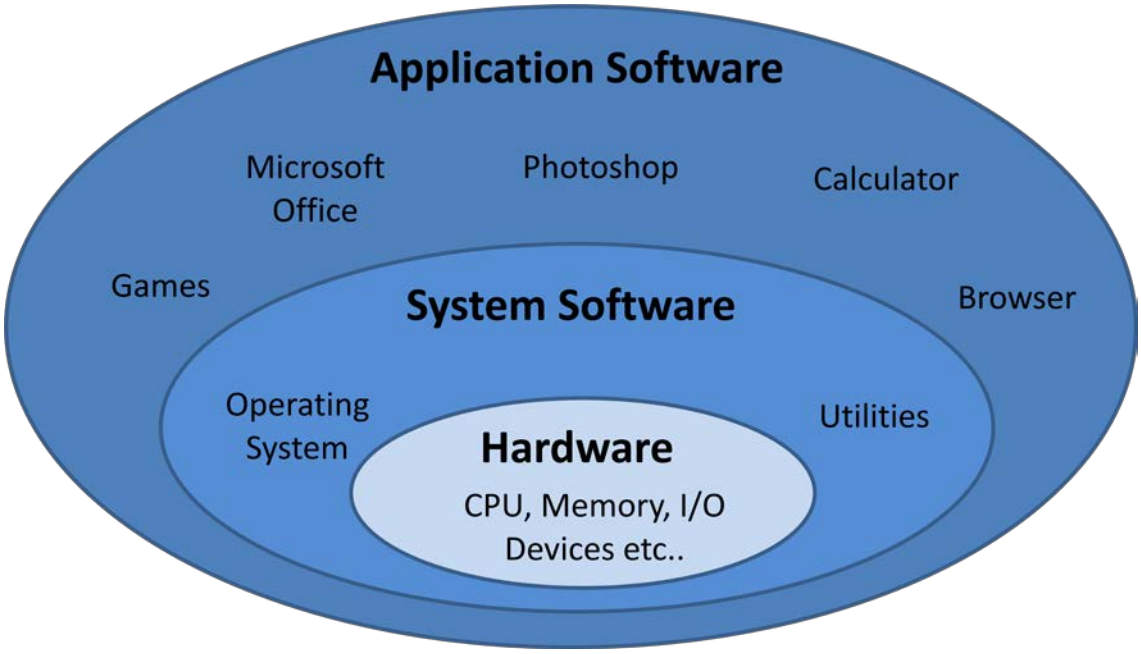
**3) System Development:** This includes various programs that help the developers to create application software. Examples: Compilers, Interpreters, Debuggers, Linkers etc.

**Application Software:** This software allows the users to use the computer for various purposes. Application software is divided into two classes: 1) General purpose software and 2) Application specific software.

**1) General Purpose Software:** This software can be used for more than one purpose. Examples: Microsoft Office Suite, Photoshop, Visual Studio etc.

**2) Application Specific Software:** This software is used only for a specific purpose. Example: Tally, Turbo C Complier etc.

The relationship between hardware, system software and application software is as shown in the below diagram:

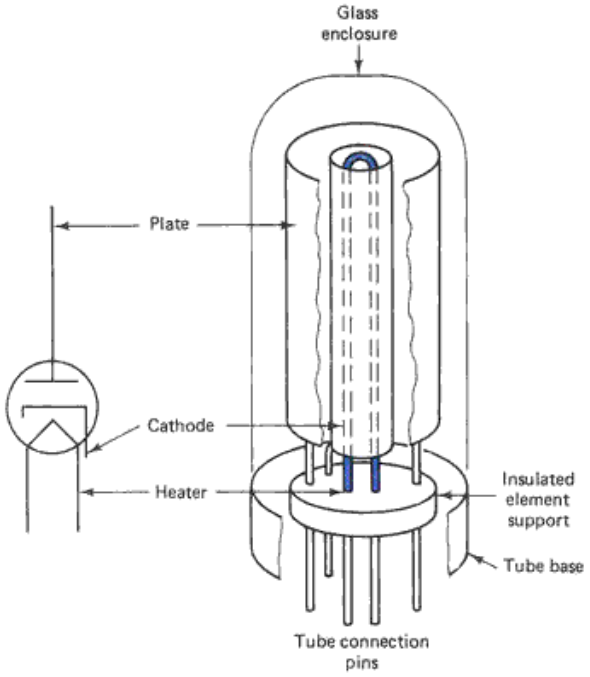


## Generations of Computers / History of Computers

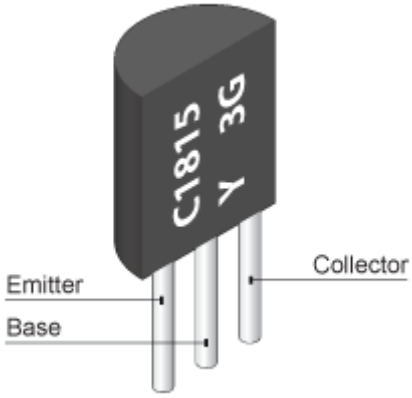
The history of computers is often referred to in terms of five distinct eras/generations. Each generation is characterized by major technological development that fundamentally changed the way computers operate, resulting in increasingly: smaller, cheaper, more powerful and more efficient devices. The five generations are:

- First Generation (1946 to 1955)
- Second Generation (1956 to 1963)
- Third Generation (1964 to 1970)
- Fourth Generation (1971 to 1991)
- Fifth Generation (1992 to Present)

**First Generation:** The first generation computers used vacuum tubes which are large in size. The size of first generation computers was usually very large which occupied an entire room in a building. These computers were very expensive and were limited in memory. Also these computers generated a lot of heat. The language used for programming was machine language. UNIVAC is a famous example for first generation computer. A vacuum tube is as shown below:

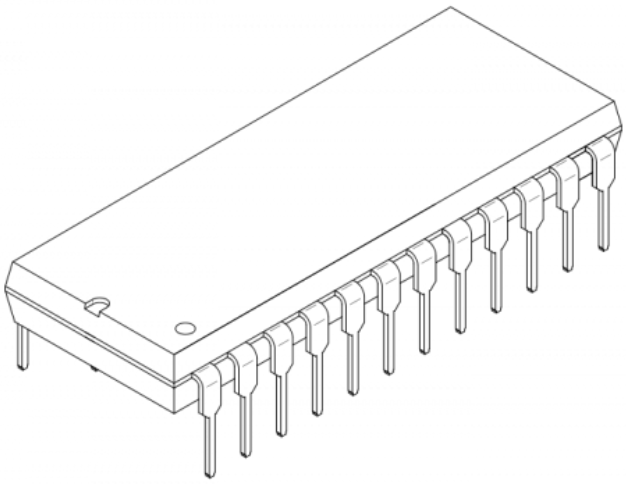



**Second Generation:** In the second generation computers, vacuum tubes are replaced by electronic component called transistor. Transistors are small in size when compared to vacuum tubes. So, the second generation computers are still huge but smaller when compared to first generation computers. Heat generation was still high. Assembly languages were used for programming. A transistor is as shown below:




**Third Generation:** In third generation computers, new electronic components called integrated circuits (IC) were used. Generally these ICs consist of thousands of transistors along with other electronic components embedded

onto a single electronic circuit board by using technology called Large Scale Integration (LSI). Third generation computer were smaller when compared to previous generation computers. They were fast and efficient. High-Level languages were used for programming. An integrated circuit chip is as shown below:



**Fourth Generation:** In fourth generation computers, ICs were replaced by microprocessors which consist of millions of transistors along with other electronic devices. These microprocessors were made by using Very Large Scale Integration (VLSI) technology. These computers are powerful, smaller and cheaper when compared to previous generation computers. A microprocessor is as shown below:


**Fifth Generation:** The fifth generation is not declared officially yet. The fifth generation is not characterized by the change is hardware technology rather it is the change in computer usage. Fifth generation computers depend on the concept called Artificial Intelligence (AI) which enables the computer to behave like a human.

## Types of Computers

There are 5 types of computers categorized based on the size, storage capacity and cost. The five types of computers are:

1) Super Computers
2) Mainframe Computers
3) Mini Computers
4) Workstations
5) Personal or Desktop Computers.

**Super Computers:** These types of computers are very large in size which normally occupies the size of entire room in a building. These computers have high data processing speeds and high storage capacity. These computers are generally used by scientific and research organizations and are used for specific data processing tasks. Examples: Belle, BlueGene, Hydra, Tianhe-1A.

**Mainframes:** These types of computers are smaller than super computers, but are still large in size. These computers are used by national and international organizations for data processing tasks. These computers have large storage capacity but lesser than super computers. Examples: IBM system 3, AS-400.

**Mini Computers:** These types of computers are close to the size of mainframes, but lesser than mainframes. These computers are used by large organizations. Examples: DEC PDP, Data General Nova.

**Workstations:** These computers look like desktop computers, but generally posses high data storage capacity and larger memory than desktop computers. These computers are used for specific tasks like: animations, game designing, CAD/CAM and other heavy data processing and graphics oriented tasks.

**Personal or Desktop Computers:** These computers are the general purpose computers that people use at their workplaces and at their house. These computers have less storage capacity and less processing speed than other types of computers. These computers are generally used for gaming, accounting, presentations etc.

## Networking Concepts

### Network

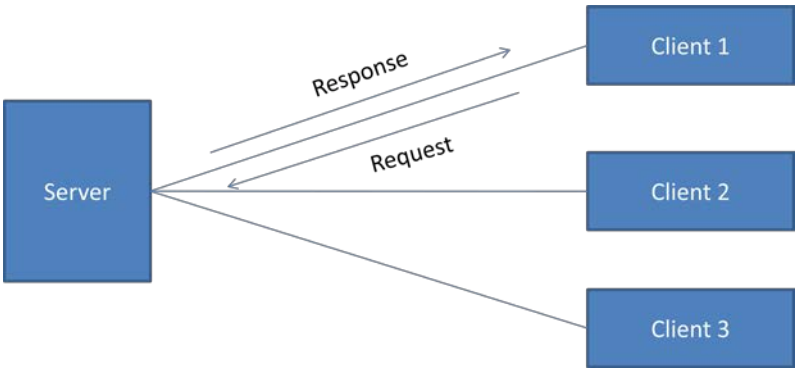
A network is a collection of interconnected computers, which are connected using communication media along with communication devices. Examples of communication devices are: modems, routers and bridges. Examples of communication media are: telephone cables, coaxial cables, twisted pair cables and fiber optical cables.

Using a network has several advantages like:

1) Easy to share the information.
2) Easy to share the resources like printers etc.
3) Saves time and money.

## Network Model

A network model refers to how the computers over the network communicate with each other. The famous networking model is client-server model. In this model there are two types of entities called client and server. A client always issues requests and the server always serves the requests. This can be viewed diagrammatically as shown below:




## Topology

The topology of a network refers to the structure in which the computers are interconnected with one another. Some of the network topologies are mesh, star, bus, ring etc.

## Internet

The collection of networks throughout the world which consists of billions of computers is known as internet. We can transfer data from one part of the world to anywhere in the world.

## WWW (World Wide Web)

The World Wide Web (WWW) is the collection of all the webpages available on the internet. The webpages consists of information in the form of text, images, audio or video. One webpage can be linked with another webpage by using hyperlinks.

# Computer Languages

To communicate with a computer, we use computer languages or also known as programming languages. To write a program, we must use the computer languages. A computer language is a set of predefined words that are combined into a program according to predefined rules (syntax). Over the years, computer languages had evolved from machine language to high-level languages. There are generally three types of computer languages. They are: 1) Machine Language, 2) Assembly Language and 3) High-level languages.

## Machine Language

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made up of streams of 0's and 1's. The only language understood by a computer is the machine language. This language is tightly coupled with the computer hardware. It is difficult to write and maintain code in machine language.

**Table 9.1** Code in machine language to add two integers

| *Hexadecimal* | *Code in machine language* | | | |
|---|---|---|---|---|
| $(1FEF)_{16}$ | 0001 | 1111 | 1110 | 1111 |
| $(240F)_{16}$ | 0010 | 0100 | 0000 | 1111 |
| $(1FEF)_{16}$ | 0001 | 1111 | 1110 | 1111 |
| $(241F)_{16}$ | 0010 | 0100 | 0001 | 1111 |
| $(1040)_{16}$ | 0001 | 0000 | 0100 | 0000 |
| $(1141)_{16}$ | 0001 | 0001 | 0100 | 0001 |
| $(3201)_{16}$ | 0011 | 0010 | 0000 | 0001 |
| $(2422)_{16}$ | 0010 | 0100 | 0010 | 0010 |
| $(1F42)_{16}$ | 0001 | 1111 | 0100 | 0010 |
| $(2FFF)_{16}$ | 0010 | 1111 | 1111 | 1111 |
| $(0000)_{16}$ | 0000 | 0000 | 0000 | 0000 |

## Assembly Language

The next evolution in programming came with the idea of replacing binary code for instruction and addresses with symbols or mnemonics. Because they used symbols, these languages were first known as symbolic languages. The set of these mnemonic languages were later referred to as assembly languages. It is easy to write and maintain programs in assembly language than in machine languages.

**Table 9.2** Code in assembly language to add two integers

| *Code in assembly language* | *Description* |
|---|---|
| **LOAD RF Keyboard** | Load from keyboard controller to register F |
| **STORE Number1 RF** | Store register F into Number1 |
| **LOAD RF Keyboard** | Load from keyboard controller to register F |
| **STORE Number2 RF** | Store register F into Number2 |
| **LOAD R0 Number1** | Load Number1 into register 0 |
| **LOAD R1 Number2** | Load Number2 into register 1 |
| **ADDI R2 R0 R1** | Add registers 0 and 1 with result in register 2 |
| **STORE Result R2** | Store register 2 into Result |
| **LOAD RF Result** | Load Result into register F |
| **STORE Monitor RF** | Store register F into monitor controller |
| **HALT** | Stop |

## High-Level Language

Although assembly languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware they were using. Working with symbolic languages was also very tedious, because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level languages.

Examples of high-level languages are:

1. BASIC (Beginners All Purpose Symbolic Instruction Code).
2. FORTRAN (Formula Translation).
3. PL/I (Programming Language, Version 1).
4. ALGOL (Algorithmic Language).
5. APL (A Programming Language).
6. COBOL (Common Business Oriented Language).
7. RPG (Report Program Generator).
8. LISP (List Processing).
9. Prolog (Program in Logic).
10. C++
11. Java
12. Visual Basic
13. C

**Program 9.1** Addition program in C++

```
/*      This program reads two integers from keyboard and prints their sum.
        Written by:
        Date:
*/
#include <iostream.h>
using namespace std;
int main (void)
{
        // Local Declarations
        int number1;
        int number2;
        int result;
        // Statements
        cin >> number1;
        cin >> number2;
        result = number1 + number2;
        cout << result;
        return 0;
} // main
```

## Translation

Programs today are normally written in one of the high-level languages. To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run. The program in a high-level language is called the source program. The translated program in machine language is called the object program. Two methods are used for translation: **compilation** and **interpretation**.

### Translator

A translator is a program, which converts the code written in one language into another language. The widely used translators are compilers and interpreters.

### Compiler

A compiler is a translator which converts the program written in high-level language into assembly code or into another form of intermediate code or directly into machine code. A compiler converts the whole source code at once into object code and then executes it. So, compiler is faster than a interpreter.

### Interpreter

An interpreter is a translator which converts the source program into object or machine code. The interpreter converts the source code line-by-line and executes it immediately, which results in less performance. Thus, an interpreter is slower than a compiler.

## Generations of Programming Languages

There are five generations of programming languages. There are classified based on how close the programming language is to human beings.

**First Generations Languages (1GL):** The first generation language is the machine language. It consists of only 0's and 1's. It is very difficult write programs in machine language.

**Second Generation Languages (2GL):** The second generation language is the assembly language. Assembly language consists of symbols known as mnemonics, english words rather than 0's and 1's. Programs written in assembly language are converted to machine language using a translator known as assembler.

**Third Generation Languages (3GL):** The third generation languages are high-level languages which are similar to english. It is easy to write programs using high-level languages. Programs written in high-level language are converted to machine language by using a translator like compiler or interpreter. Third generation languages are problem oriented languages. Examples: C, FORTRAN, COBOL, PASCAL etc.

**Fourth Generation Languages (4GL):** The fourth generation languages are non-procedural languages. Programmers have to specify only what to do but not how to do it. These languages are developed for users having minimum programming language. Examples: SQL, ABAP etc.

**Fifth Generation Languages (5GL):** The fifth generation languages are declarative languages which are used in artificial intelligence and expert systems. Example: Prolog etc.

## Steps in Program Development

A computer program is a set of formal instructions, which the computer executes in order to carry out some designated task. Whether that task is as simple as adding two numbers together or as complex as maintaining a large inventory for a multi-national corporation, there is a common element involved. They are both achieved by the computer executing a series of instructions – *the computer program*.

Programming can be defined as the development of a solution to an identified problem. There are six basic steps in the development of a program:

**1. Define the problem**

This step (often overlooked) involves the careful reading and re-reading of the problem until the programmer understands completely what is required.

**2. Outline the solution (analysis)**

Once the problem has been defined, the programmer may decide to break the problem up into smaller tasks or steps, and several solutions may be considered. The solution outline often takes the shape of a hierarchy or structure chart.

**3. Develop the outline into an algorithm (design)**

Using the solution outline developed in step 2, the programmer then expands this into a set of precise steps (algorithm) that describe exactly the tasks to be performed and the order in which they are to be carried out. This step can use both structured programming techniques and pseudocode.

**4. Code the algorithm into a specific programming language (coding)**

It is only after all design considerations have been met that a programmer should actually start to code the program. In preceding analysis it may have been necessary to consider which language should be used, as each has its own peculiarities (advantages and disadvantages).

**5. Run the program on the computer (testing)**

This step uses a program compiler and test data to test the code for both syntax and logic errors. If the program is well designed then the usual time-wasting frustration and despair often associated with program testing are reduced to a minimum. This step will often need to be done several times until the programmer is satisfied that the program is running as required.

**6. Document and maintain the program (documentation)**

Program documentation should not be just listed as the last step in the development process, as it is an ongoing task from the initial definition of the problem to the final test results. Documentation also involves maintenance - the changes that are made to a program, often by another programmer, during the life of that program. The better a program has been documented and the logic understood, the easier it is for another to make changes.

# Algorithm

## Definition

An algorithm is a finite sequence of step by step, discrete, unambiguous instructions for solving a particular problem.

Or

Algorithm is a step-by-step procedure for solving a particular problem.

Or

An algorithm is an ordered set of unambiguous executable steps, defining a terminating process.

An algorithm, named after 9th centaury Persian mathematician al-Khowarazmi, is simply a set of rules for carrying out some calculation, either by hand, or, more usually, on a machine. An algorithm is like a recipe. It lists the steps involved in accomplishing a task. It can be defined in programming terms as a set of detailed, unambiguous and ordered instructions developed to describe the processes necessary to produce the desired output from the given input. The algorithm is written in simple English and is not a formalized procedure, however to be useful there are some principles which should be adhered to. An algorithm must:

1. Be precise and unambiguous
2. Give the correct solution in all cases and
3. Eventually end.

Algorithms can be represented in different ways. They are:

1) Natural languages
2) Pseudocode
3) Flowcharts
4) Control tables

## Pseudocode

Pseudocode is easy to read and write, as it represents the statements of an algorithm in English. Pseudocode is really structured English. It is English which has been formalised and abbreviated to look very like a high-level computer language.

There is currently no standard pseudocode. Different textbook authors seem to adopt their own special techniques and set of rules, which often resemble a particular programming language. But as a guideline, try the following conventions:

1. Statements are written in simple English.
2. Each instruction is written on a separate line.
3. Keywords and indentation are used to signify particular control structures.
4. Groups of statements may be formed into modules and that group given a name.

If a programmer uses words and phrases in his or her pseudocode that are in line with basic computer operations, then the translation from the pseudocode algorithm to a specific programming language becomes quite simple. We could break down the computers operations into six basic operations and represent these operations in pseudocode. Each operation can be represented as an English instruction with keywords and indentation to signify a particular control structure:

**1. A computer can receive information** – when a computer is required to receive information or input from a particular source, whether it is a keyboard, disk or other source, the verbs *Read* and *Get* can be used

e.g. Read student name
Get system date
Read number_1, number_2

**2. A computer can put out information** – when a computer is required to supply information or output to a device the verbs *Print*, *Write* or *Put* could be used

e.g. Print student number
Write 'Program completed'
Write customer record to master file
Put out name, address, postcode

**3. A computer can perform arithmetic** – most programs require the computer to perform some sort of mathematical calculation, or formula. You could use actual mathematical symbols or the words for those symbols

e.g. Add Number to Total
Total = Total + Number
Divide Totalmarks by Studentcount
SalesTax = Costprice * 0.10
Compute C = (F – 32) * 5/9

**4. A computer can assign a value to a piece of data** – there are many cases where a programmer will need to assign a value to a piece of information. Firstly to give data an initial value in pseudocode the verbs, *Initialise* or *Set* can be used. Secondly to assign a value as a result of some processing, the symbol "=" is written. Thirdly, to keep a piece of information for later use the verbs *Save* or *Store* could be used

e.g. Initialise total accumulators to zero
Set StudentCount to 0
TotalPrice = CostPrice + SalesTax
Store CustomerNumber in LastCustomerNumber

**5. A computer can compare two pieces of information and select one of two alternatives** – an important ability is the comparison of two pieces of information and then as a result of the comparison select on of two alternate actions. To represent this operation in pseudocode the keywords IF, THEN and ELSE can be used

```
e.g.  IF student is parttime THEN
            Add 1 to parttimecount
      ELSE
            Add 1 to fulltimecount
      ENDIF
```

Note the use of indentation to emphasize the THEN and ELSE options and the use of the delimiter ENDIF to close the operation.

**6. A computer can repeat a group of actions** – when there is a sequence of steps which need to be repeated, the keywords DOWHILE and ENDDO could be used

```
e.g.  DOWHILE studenttotal < 50
            Read student record
            Write student name, address to report
            Add 1 to Studenttotal
      ENDDO
```

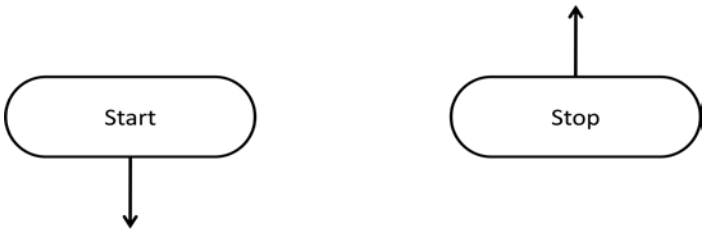
Once again note the use of indentation. It is easy to read where the loop starts and ends. Alternatives to the above could be WHILE and ENDWHILE.

## Flowchart

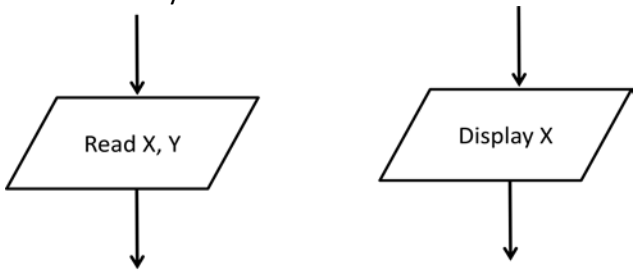
A flowchart is a diagrammatic or pictorial or visual representation of an algorithm. It is a diagram which consists of various graphic symbols for representing the nature and flow of steps in a program or process.
The fundamental or basic elements in a flowchart are:

1. Start/Stop symbol
2. Input/output symbol
3. Process symbol
4. Decision or Test symbol
5. Loop symbol
6. Connector symbol

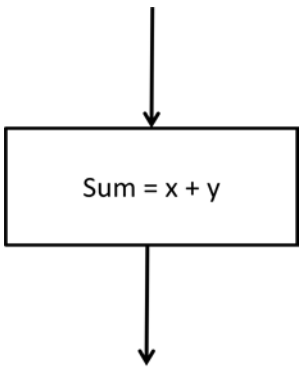
**Start/Stop symbol:** The start and end symbols indicate the beginning and the end of the flowchart. This symbol looks like a flat oval. Only one flow line is combined with this kind of symbol. Generally this symbol is used twice in a flowchart, that is, at the beginning and at the end. The start/stop symbol is as shown below:



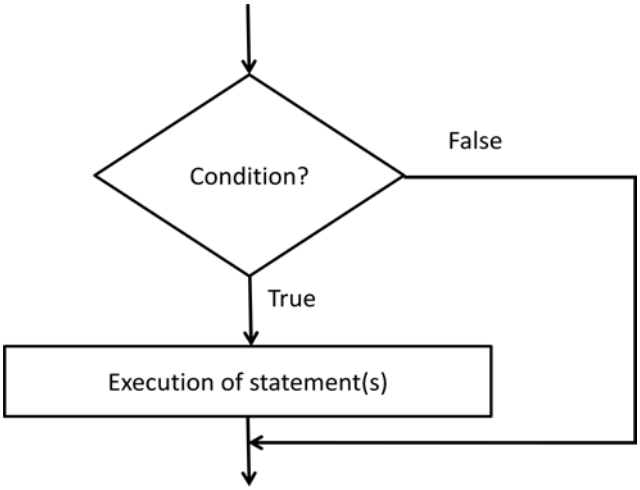

**Input/output symbol:** The input/output symbol looks like a parallelogram as shown below. This symbol is used to input and output the data. There are two flow lines connected with the input/output symbol. One line comes to this symbol and the other line goes out from this symbol.



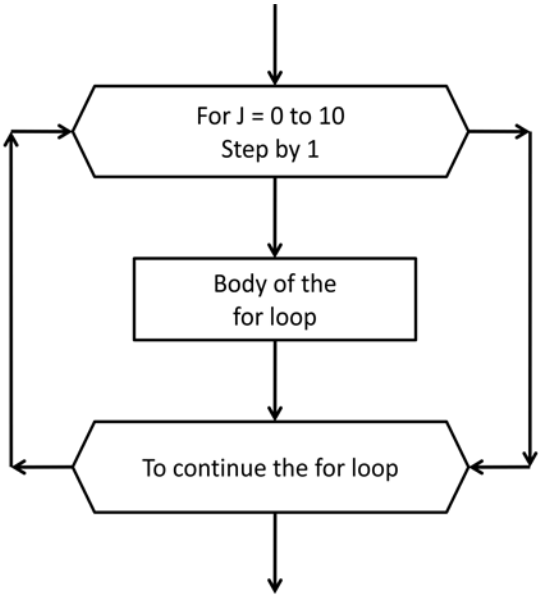

**Process Symbol:** The process symbol looks like a rectangle. It is used for data processing and assigning values to variables. The operations mentioned within the rectangular block will be executed when this kind of block is entered in the flowchart. There are two flow lines associated with this symbol. The process symbol is as shown below:



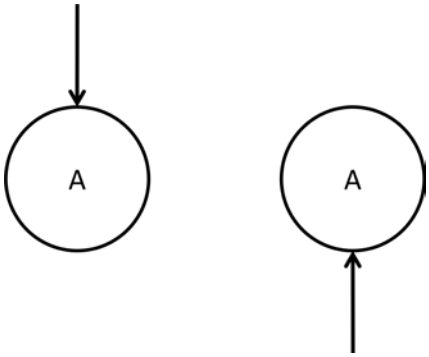

**Decision/Test symbol:** The decision symbol is like a diamond/rhombus. This symbol is used to take one of the decisions. Depending on the condition the decision block selects one of the alternatives. While solving a problem, one can take a single alternative or two or multiple alternatives, depending on the situation. A single alternative decision symbol is shown below:




**Loop Symbol:** This symbol looks like a hexagon. This symbol is used for implementing loops. Four flow lines are associated with this symbol. Two lines are used to indicate the sequence of the program and the remaining two are used to show the looping area, that is from the beginning to the end. Loop symbol is as shown below:




**Connector Symbol:** The connector symbol is a circle. It is used to establish the connection whenever it is impossible to directly join two parts in a flowchart. Quite often, the two parts of the flowchart may be on two separate pages. In such case, a connector can be used for joining the two parts. Only one flow line is associated with this symbol. It is as shown below:



## Examples:

**Problem: Write an algorithm for adding two integers.**

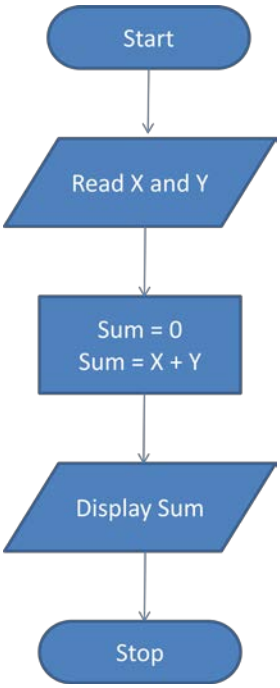
### Algorithm

Algorithm: Add(x,y)
To add two integer x and y
Inputs: x and y
Output: Sum of x and y
Start

1: Read two integers x and y
2: Set sum = 0
3: sum = x + y
4: Display sum

Stop

### Flowchart



Problem: Write an algorithm to find the greatest number among two integers.

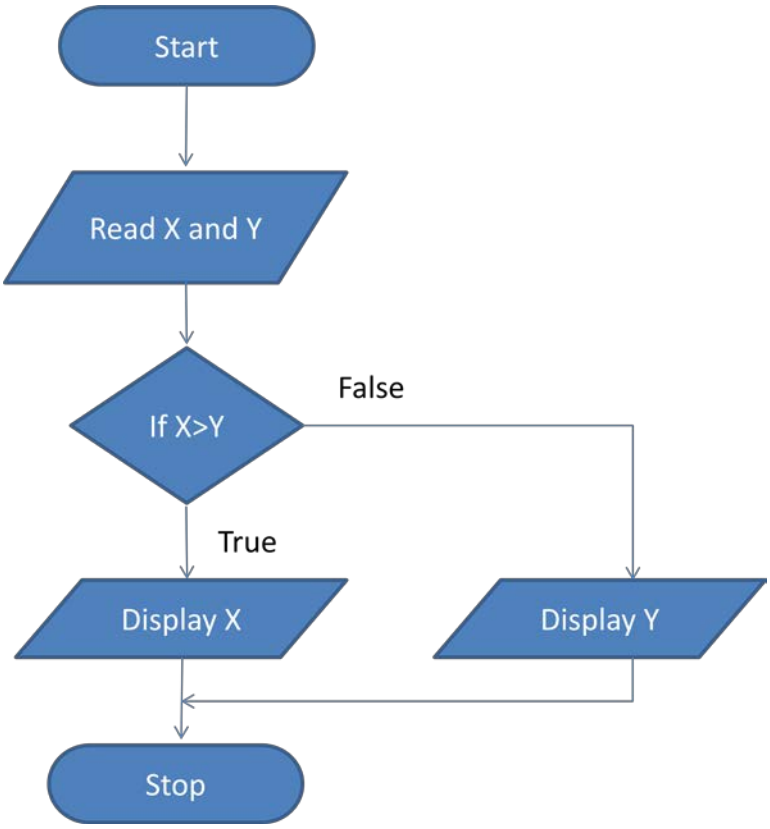
## Algorithm

Algorithm: Greatest(x,y)
To find the greatest among two integers
Inputs: x and y
Output: Greatest integer among x and y
Start

```
1:      Read two integers x and y
2:      IF x is greater than y THEN:
            2.1:    Display "X is greater"
        ELSE
            2.1:    Display "Y is greater"
        END IF
```

Stop

## Flowchart



# Introduction to C

## Basic Programming Terminology

### Language

A language is a set of characters, words and associated grammar rules which is used for communication between two human beings either spoken or written. Examples: Telugu, Hindi, English etc.

### Programming Language

A language which is used for communication between a human being and a computer. Human beings communicate with the computer by writing programs using a programming language. Ex: C, C++, Java etc.

### Program

A program is a set of instructions for solving a particular problem or let the computer perform operations for completing a task. Programs are written using programming languages.

### Programming

The process of writing programs to instruct the computer do something is known as programming. Programming is done using programming languages.

### Programmer

A person who can write programs by using any one of the programming languages is known as a programmer. A programmer has the knowledge of using atleast one programming language for writing programs.

### Software

Software is a collection of programs. Generally software is developed for solving complex problems. A person who develops software is known as a software developer.

## What is C?

C is a structured mid-level programming language for developing software. C follows structured programming in which the instructions are arranged or structured using blocks and functions. C is used for developing operating systems which are system software and also used to create application software. Hence, C is known as middle level language.

## Why C? (Advantages of C)

C is a powerful language using which we can develop operating systems, games and other application software. The advantages or characteristics of C language are:

**1) Powerful and Flexible:** C is a flexible programming language. Using the concepts in C, a programmer can develop software for almost any kind of purpose. Various concepts like pointers, function, structures in C make it a powerful language.

**2) Popular:** C has wide spread popularity in the programming league as it is very easy to learn, easy to write programs and easy to understand the programs written in C.

**3) Portable:** C is a machine independent programming language. Code written in C, can be executed on any machine irrespective of the underlying hardware with little or no modifications.

**4) Minimum set of Keywords:** C89 has 32 keywords. These small set of keywords add power to C language. A large set of keywords does not mean that the language is more powerful.

**5) Modular:** In C, we can divide a problem into sub problems and solve the sub problems individually using the concept called as functions. This makes the programs written in C modular.

**6) C and C++:** As C++ is a superset of C, it inherits all the concepts in C, to which other new concepts have been added. So, you can know how powerful the C language is.

## Applications of C

The applications which can be developed using C language are literally unlimited. Some of the types of applications which can be developed using C language are:

- Operating Systems
- Games
- Utilities
- Drivers
- Graphics
- Embedded System Software

## History of C

The root of all modern languages is ALGOL (Algorithmic Language), developed in the early 1960's. ALGOL was the first language which introduced the concept of structured programming. In 1967, Martin Richards developed the language called BCPL (Basic Combined Programming Language) for writing system software. In 1970, Ken Thompson created a new language using the features from BCPL and called it B. Both BCPL and B were typeless system programming languages.

C was developed by Dennis Ritchie in 1972 by using the features of ALGOL, BCPL and B programming languages along with new features like data types. For many years, C was used mainly in academic institutions, but with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

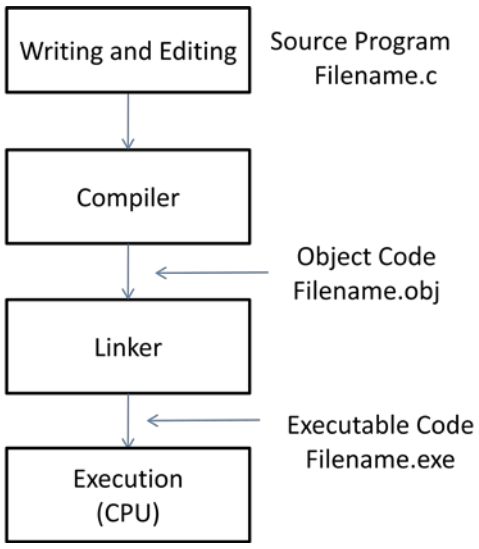
The language became more popular after publication of the book, "The C Programming Language" by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C". The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This was a serious problem for system developers.

In 1983, ANSI (American National Standards Institute) appointed a technical committee to define a standard for C. The committee a version of C in 1989 which is now known as ANSI C. It was then later in 1990 approved by ISO (International Standards Organization). This version of C is also referred as C89.

In 1999, the standardization committee of C has added some new features to enhance the usefulness of the language. The result was the 1999 standard for C, which also known as C99. In 2011, some more new features are added to the C language, which is known as C11.

## Creating and Running C Programs

The steps for creating and running programs are: writing/editing, compiling, linking and execution. This can be viewed diagrammatically as shown below:



**Writing/Editing:** The first step in creating programs is, writing or editing the program. A program can be written in any text editor like notepad. After writing a program, the program must be saved, In C language, the program is saved with the extension ".c". This is the source program written in a high-level language.

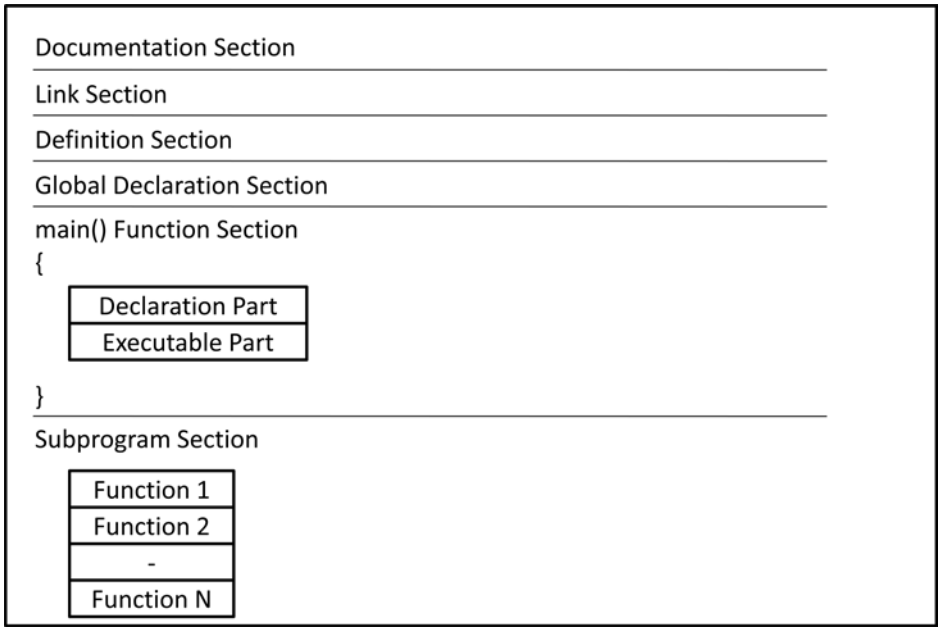
**Compilation:** After writing and saving the source program, the next step is compilation. Here we will use a software called as compiler, which converts a program written in high-level language into machine language. The resultant file is known as an object file in C. The extension of that file is ".obj".

**Linking:** After compilation the next step is linking. Here software called linker is used. The linker links the program with external library files which contains the code for predefined functions and creates an executable file. The extension of the executable file is ".exe".

**Execution:** Finally after the executable file is created after linking, the next step is execution. The operating system executes the executable file which is the machine code with the help of the CPU and other hardware components.

## Structure of a C Program

Every program written in C follows a certain structure. That structure is as shown below:




**Documentation Section:**

The documentation section is used to improve the readability and to understand various elements in the program. This section is used to provide help for the general users. This section consists of plain text written in English and includes information like author, purpose of the program, date on which the program was created etc. This section is included inside comments. Example:

```
/*********************************
 * Author: Name                  *
 * Date: 7/10/2012               *
 * Purpose: Theme of the program *
 ********************************* */
```

**Link Section:**

The link section is used to include the external header files in which the pre-defined functions are available. To use the pre-defined functions in our C programs, these files must be linked with our program. Files can be included by using the "include" directive. Example:

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

In the above example "stdio.h" is a header file which is available in the system area in the programmer's computer. The angular brackets "< and >" tells the compiler to search the file "stdio.h" in the system area. If we specify the above example as #include"stdio.h", then the compiler searches for the file "stdio.h" in the current directory.

**Definition Section:**

This section is used to define symbolic constants. Symbolic constants can be declared by using the #define directive. Example:

```
#define PI 3.142
```

**Global Declaration Section:**

This section is used for declaring global variables which can be throughout the program and for declaring function prototypes. The global variables can be accessed by any function within the file.

**main() Function Section:**

The main() function is necessary in every C program. It specifies the starting point of execution in a program. There should be a main() declared in every C program. The main() function consists of the executable instructions and declaration statements. It may call other user defined functions.

**Subprogram Section:**

This section includes the definitions of user-defined functions. A function is essentially a set of instructions for performing a task. We can divide a problem into sub problems and write a function for solving each sub problem.

**Example: HelloWorld Program**

```
/*******************************
 * Author: Teja                *
 * Date: 7/10/2012             *
 * Purpose: To print Hello World *
 ***************************** *** */
#include<stdio.h>
#include<conio.h>
main()
{
        clrscr();
        printf("HelloWorld!");
        getch();
}
```

In the above example we can see three sections: documentation section, link section and main() function section.

In the above program, **/* and */ denotes multi-line comments**. The /* indicates the starting of multi-line comment and the */ indicates the ending of multi-line comments. Everything included between /* and */ is ignored by the compiler. In the next line after the multi-line comments, **#include** is a pre-processor directive which is used to link the program with external files like header files in which the predefined functions are declared. The **main()** function is essential in every C program, as it specifies the starting point of execution in a program. Everything starting from **{** to **}** is known as the body of the main function. **clrscr()** function is used to clear the output screen (console). The **printf()** function is used to output characters onto the console. The **getch()** function pauses the execution of the program, and waits for a single character input from the user. The declaration of printf() is available in **stdio.h** (**St**andar**d** **I**nput **O**utput) header file and the declarations of clrscr() and getch() are available in **conio.h** (**Con**sole **I**nput **O**utput) header file.

# C Tokens

## Introduction

A programming language is designed to help process certain kinds of data consisting of numbers, characters and strings and to provide useful output known as information. The task of processing data is achieved by writing instructions. These set of instructions is known as a program. Programs are written using words and symbols according to the rigid rules of the programming language known as syntax.

## Character Set

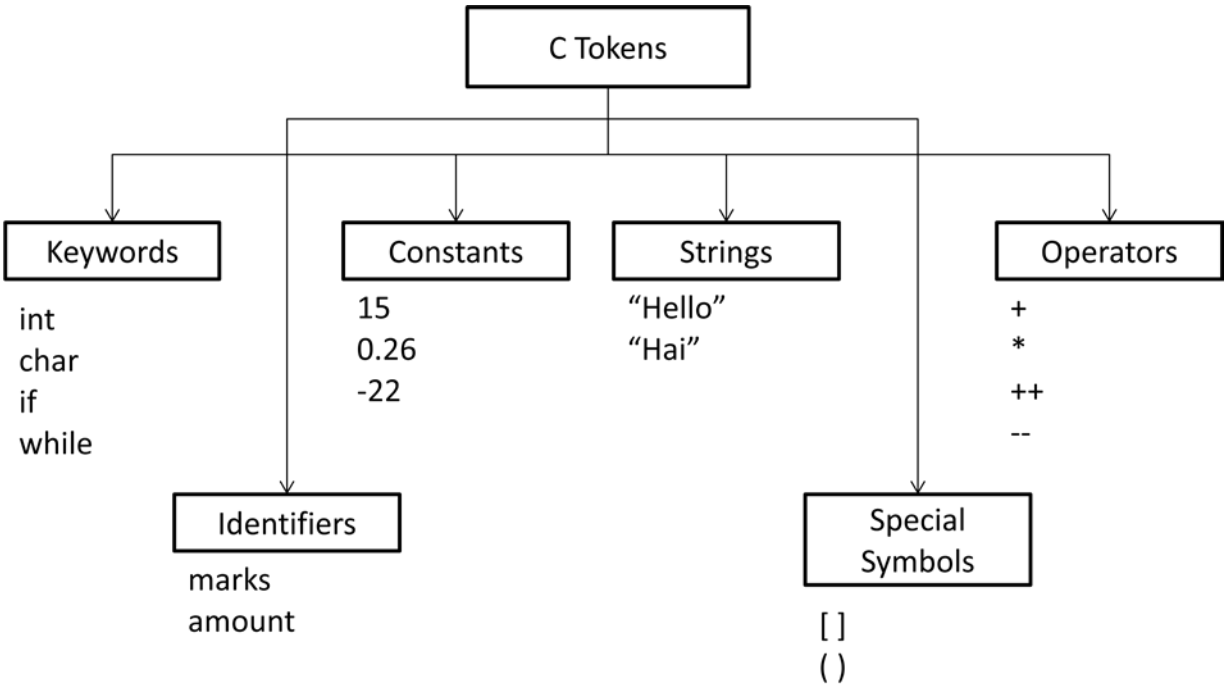
The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The characters in C, are grouped into the following four categories:

1) Letters
2) Digits
3) Special Characters
4) Whitespaces

The compiler ignores the white spaces unless they are part of the string constants. White spaces are used to separate words from each other, but they cannot be used in between the characters of keywords and identifiers.

## C Tokens

In a paragraph of text, individual words and punctuation marks are considered as tokens. Likewise in a C program, the smallest individual units are known as C tokens. C has six types of tokens as shown below. C programs are written using these tokens and the syntax of the language.




### Keywords

Every word used in a C program is classified as either a keyword or as an identifier. A keyword in C is a reserved word which has a specific meaning. Keywords in C cannot be used as identifiers. Keywords serve as the basic building blocks for program statements. Keywords in C are always in lowercase. ANSI C supports 32 keywords which are listed below:

| | | | |
|---|---|---|---|
| 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 |

## Identifiers

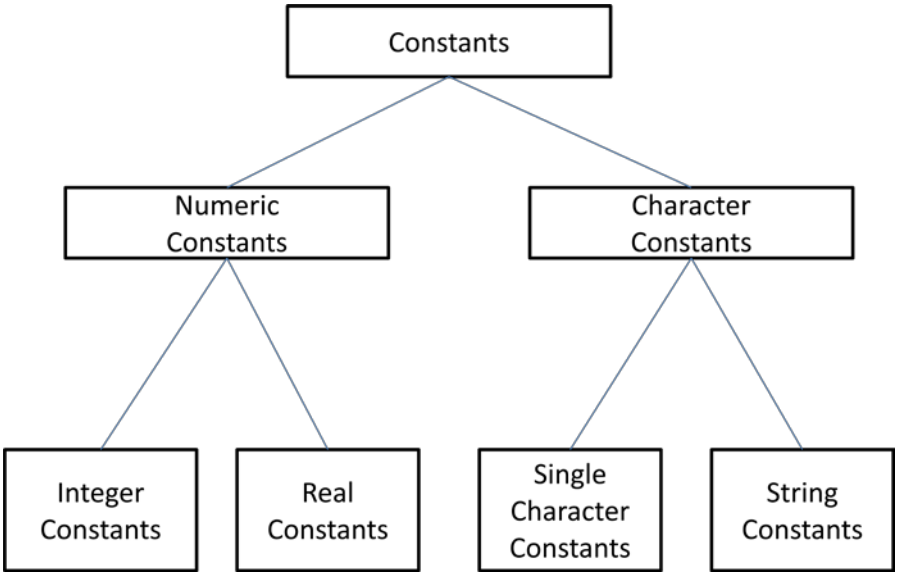
Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters can be used, although lowercase letters are generally used. The underscore character is also permitted in identifiers. There certain rules while writing identifiers. They are as follows:

1) First character must be an alphabet or underscore.
2) Must consist of only letters, digits or underscore.
3) Only first 31 characters are significant.
4) Cannot use a keyword.
5) Must not contain white space.

# Constants, Variables and Data Types

## Constants

Constants are fixed values, which do not change during the execution of a program. C supports several types of constants, which are as shown below:




### *Integer Constants*

An integer constant refers to a sequence of digits. Generally in programs, the number systems used are: decimal, octal and hexadecimal. Decimal integers consists of umbers from 0 to 9, preceded by an optional + or – sign. Some valid examples of decimal integer constants are: 133, +45, -15, 0, 342332 etc. Embedded spaces and non-digit characters are cannot be used between digits.

An octal integer consists of numbers from 0 to 7. Octal integer numbers are always preceded by a zero. For example if we have to represent 35 in octal system, we must write it as 035 in the program. Some valid examples of octal integer constants are: 035, 02, 0435, +025 etc.

An hexadecimal integer consists of numbers from 0 to 9 and 10, 11, 12, 13, 14 and 15 are represented as A, B, C, D, E and F. Hexadecimal numbers are always preceded by 0x or 0X. Some valid examples of hexadecimal integer constants are: 0x54, 0X23, +0x1F, -0X56 etc.

### *Real Constants*

Integer numbers are not sufficient to represent quantities that vary continuously, such as distances, height, prices etc. These quantities are represented by numbers containing fractional parts like 25.234. Such numbers are called as real or floating point constants. Some valid examples of real constants are: 0.067, -12.5, +4.67, .87, 121. Etc.

A real number may also be expressed in exponential (scientific) notation. For example, 45.2344 can be written 0.452344e2. The exponential notation is: ***mantissa e exponent.*** The mantissa is either a real number expressed in decimal notation or an integer with an optional + or – sign. The exponent is an integer number with an optional + or – sign. Some valid examples of real constants in exponential notation are: 0.34e2, 13e-2, -1.23e-1 etc.

### *Single Character Constants*

A single character constant or character constant contains a single character enclosed in between single quotes. Some valid examples of character constants are: 'f', 'A', '/', ';', ' ', '4'. The character constant '4' is not equal to the number 4. Every character constant will have an associated integer value known as ASCII code. Since each character constant has an associated integer value, we can perform arithmetic operations on them.

### *String Literals or String Constants*

A sequence of characters that are enclosed between double quotes is known as a string literal or string constant. The characters in a string literal can be either letters, digits or special symbols or white spaces. The string literal does not have an associated integer value like the character constants. Some valid examples of string constants are: "hai", "hEllO", "hi5", "Wel come" etc.

### *Escape Sequences*

C supports some special backslash character constants that are used in output functions like printf(). For example to move the cursor from the current line to next line, there is no standard way to achieve it. So, for such special cases, C provides escape sequences. In this case \n is used to move the cursor to next new line. Even though the escape sequences consist of a backslash (\) and a character or symbol, it is treated as a single character. C supports the following escape sequences:

| Constant | Meaning |
|---|---|
| \a | Alert |
| \b | Backspace |
| \f | Form feed |
| \n | New line |
| \r | Carriage Return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \' | Single quote |
| \" | Double quote |
| \? | Question mark |
| \\ | Backslash |
| \0 | Null |

## Variables

In the programs, generally we need to store values in the memory and perform operations on those values. This can be achieved in C, by the concept of variables. A variable is a placeholder for holding a value in the main memory (RAM). As the name implies, the value in the variable can change at any point of execution of the program. For using variables in our programs, there are essentially two steps:

1) Declare the variable

2) Initialize the variable

### *Declaring a Variable*

Before using a variable in the program, we have to declare the variable. The syntax for declaring a variable in a program is as shown below:

```
type variable-name;
```

The "type" in the above syntax represents the data type. The "variable-name" is the identifier. There are certain rules that must be followed while writing the variable name. They are as follows:

1. A variable name must always start with an alphabet (letter) or an underscore ( _ ).
2. The variable name must not be more than 31 characters. The suggested length of a variable name is 8 characters.
3. C is case sensitive. So, the variable name "average" is different from "AVERAGE".
4. Keywords must not be used for declaring variables.
5. White spaces are not allowed within the variable name.

### *Initializing a Variable*

After declaring the variable, we can assign a value to the variable. This process of assigning a value to the variable is known as initialization. Syntax for initializing a variable is as shown below:

```
variable-name = value;
```

The value we assign to the variable depends on the data type of the variable.

Example:

```
int a;

a = 10;
```

The declaration and initialization can be combined into a single line as shown below:

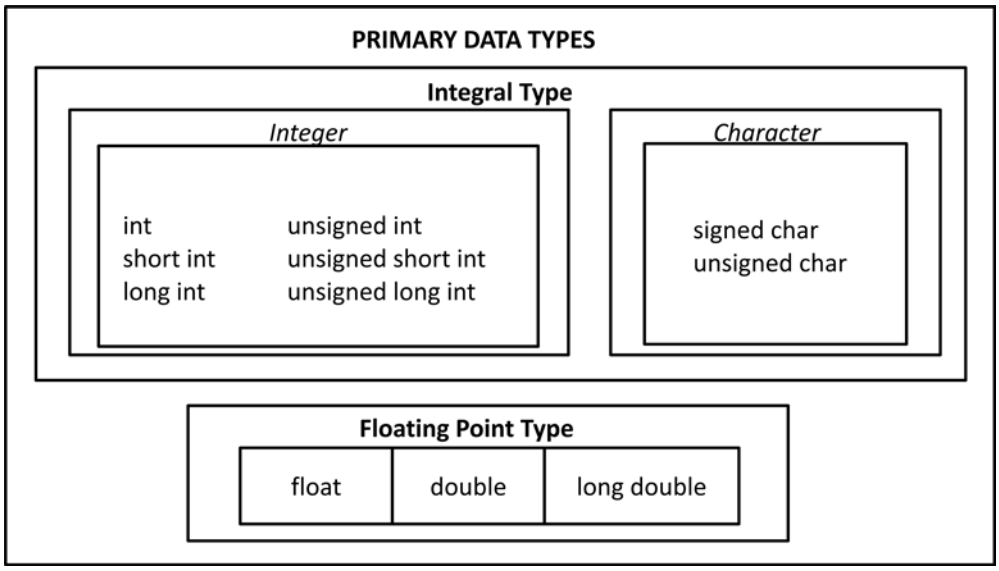
```
int a = 10;
```

# Data Types

A data type specifies the type of value that we use in our programs. A data type is generally specified when declaring variables, arrays, functions etc. In ANSI C, the data types are divided into three categories. They are:
1) Primitive or Fundamental data types
2) User-defined data types
3) Derived data types

## Primitive or Fundamental data types

The primitive data types in ANSI C are as shown in the below diagram:




The most fundamental data types that every C compiler supports are: int, char, float and double. Other compilers support the extended versions of these fundamental data types like: short int, long int, long double etc. The signed data types are used for storing both positive and negative values where as the unsigned data types are used to store only positive values. The memory capacity of these data types are based on the hardware. The memory capacity of all the fundamental data types on a 16-bit machine is as shown below:

| Type | Size (bits) | Size (bytes) | Range |
|---|---|---|---|
| char | 8 | 1 | -128 to 127 |
| unsigned char | 8 | 1 | 0 to 255 |
| int | 16 | 2 | $-2^{15}$ to $2^{15}-1$ |
| unsigned int | 16 | 2 | 0 to $2^{16}-1$ |
| short int | 8 | 1 | -128 to 127 |
| unsigned short int | 8 | 1 | 0 to 255 |
| long int | 32 | 4 | $-2^{31}$ to $2^{31}-1$ |
| unsigned long int | 32 | 4 | 0 to $2^{32}-1$ |
| float | 32 | 4 | 3.4E-38 to 3.4E+38 |
| double | 64 | 8 | 1.7E-308 to 1.7E+308 |
| long double | 80 | 10 | 3.4E-4932 to 1.1E+4932 |

## User-Defined data types

ANSI C allows the users to define identifiers as their own data types, based on the already existing primitive or fundamental data types. This concept is known as "type definition" and the data types thus created are known as user-defined data types. We can create user-defined data types in two ways:

1) By using the "typedef" keyword
2) By using the "enum" keyword

### *typedef Keyword*

The "typedef" keyword can be used to declare an identifier as a user-defined data type. The syntax for using typedef is as shown below:

```
typedef int rollno;

typedef float average;
```

In the above example, rollno and average are new user-defined data types. Now we can use rollno and average as data types as shown below:

```
rollno r1, r2;

average a1, a2;
```

### *enum Keyword*

The "enum" is used to declare identifiers as user-defined data types. Such data types are also called as enumerations. The "enum" keyword can be used to declare an identifier as data type which can store a limited set of integer values. The syntax of using "enum" keyword is as shown below:

```
enum identifier {value1, value2,....., value n};
```

Example usage of enum is as shown below:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat};
```

In the above example, "days" is a new user-defined type. All the variables created using the "days" data type can only contain values in the range Mon to Sat as shown in the above example. We can declare variables by using the "days" data type as shown below:

```
enum days day1 = Mon;
```

If we display "day1" in the printf() statement, it will display it as zero. Generally, the value of a first element in an enumeration always starts with zero and the next element's value is previous element's value plus 1. So, the value of "Tue" will be one and so on.

## Derived data types

The data types which are created using the already existing primitive or fundamental types are known as derived data types. Like user-defined data types we cannot declare new variables using the derived data types. Examples of derived data types in C are: arrays, functions, structures, unions and pointers.

# Constants

Constants in C are fixed values which cannot be changed during the execution of the program. In C, we can declare variables as constants in two ways. They are:

1) By using “const” keyword
2) By using “#define” preprocessor directive

## *const Keyword*

We can declare a variable as a constant by using the **const** qualifier. A variable that is declared using **const** cannot change its value once it is initialized. The syntax of using the **const** is as shown below:

```
const type variable-name = value;
```

Example of using the **const** keyword is as shown below:

```
const int max = 200;
```

## *#define Preprocessor Directive*

We can also declare constants in a program by using the **#define** preprocessor directive. Such constants declared using the **#define** preprocessor directive are known as **symbolic constants.** The syntax of using **#define** is as shown below:

```
#define NAME value
```

Example of using **#define** is shown below:

```
#define PI 3.142
```

Take care that there is no semi-colon (;) at the end of **#define** statement. In the above example we have assigned the value 3.142 to the symbol PI. Now, whenever PI is encountered in the program, it will be replaced with the value 3.142.

Following are the rules that apply to a **#define** statement:

1) All rules that apply to variables also apply to symbolic name. Generally symbolic names are written in CAPTIAL LETTERS, to distinguish them from the normal variable names.
2) No blank space between # and define is allowed.
3) # must be the first character in the **#define** statement.
4) A blank space is required between **#define** and the symbolic name and between symbolic name and the value.
5) **#define** statements must not end with a semi-colon.
6) After declaring a symbolic constant, we must not use it in an assignment statement.
7) Symbolic names are not declared for data types. Its data type depends on the type of the constant value.
8) **#define** statements may appear anywhere in the program, but before they are referenced in the program. Generally they are placed at the top of the program after the **#include** statements.

## *const Vs #define*

The purpose of both **const** and **#define** is to declare constants in a C program. The difference between them is: when we declare a variable as a constant using **const**, it is still treated as a variable but the value will be fixed and cannot be changed. But, when we use **#define,** we are declaring symbolic constants, which are not treated as variables. Instead the compiler searches for the symbol throughout the program and replaces the symbol with the value. Another difference is, **const** is a keyword, where as **#define** is a preprocessor directive.

## volatile Keyword

ANSI standard defines another qualifier **volatile** that could be used to tell the compiler that a variable's value may be changed at any time by some external sources (from outside the program). Example usage of the **volatile** keyword is shown below:

```
volatile int date;
```

The value of date can be modified by its own program too. If you don't want the program to modify the value of date variable, but it can be modified by external factors, then date can be declared as both **volatile** and **const** as shown below:

```
volatile const int date = 21;
```

## Reading input from Keyboard

Another way of assigning values to variables besides directly assigning the value is reading values from the keyboard. C provides **scanf** function in the **stdio.h** header file. Using this function we can read values from the keyboard and assign the values to variables. The syntax of **scanf** function is as shown below:

```
scanf("control string",&variable1,&variable2,...);
```

The **control string** specifies the type of value to read from the keyboard and the ampersand symbol **&** is an operator to specify the address the variable(s). Example usage of **scanf** function is shown below:

```
scanf("%d", &var);
```

## ASCII (American Standard Code for Information Interchange) Code

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | Null | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | Start of heading | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | Start of text | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | End of text | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | End of transmit | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | Enquiry | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | Acknowledge | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | Audible bell | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | Backspace | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | Horizontal tab | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage return | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data link escape | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg. acknowledge | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End trans. block | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitution | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | File separator | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | Group separator | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | Record separator | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | □ |

## Format Specifiers used in printf()

| Data Type | Format Specifier |
|---|---|
| int | %d (or) %i |
| unsigned int | %u |
| short int | %hd |
| unsigned short int | %hu |
| long int | %ld |
| unsigned long int | %lu |
| float | %f (or) %e (or) %g |
| double | %f (or) %e (or) %g |
| long double | %Lf |
| char | %c |
| unsigned char | %c |
| string | %s |

## Examples

**/* C program to illustrate integer constants */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        clrscr();
        printf("%d \n",20);
        printf("%d \n",-55);
        printf("%u \n",65535U);
        printf("%ld \n",999999999L);
        printf("%o \n",040);
        printf("%x \n",0x23);
        getch();
}
```

Explanation:
In the above c program, 20, -55, 65535 and 999999999 represent decimal integer constants. Whereas 040 represents octal integer constant and 0x23 represents hexadecimal integer constant. %d is the format specifier for displaying decimal integers, %u for displaying unsigned decimal integers, %ld for displaying long decimal integers, %o for displaying octal integers and %x or %X for displaying hexadecimal integer values.

**/* C program to illustrate real constants */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        clrscr();
        printf("%f \n",1.5232);
        printf("%f \n",-45.2f);
        printf("%f \n",32.);
        printf("%f \n",.56);
        printf("%e \n",2.3e2);
        getch();
}
```

Explanation:
In the above program, 1.5232, -45.2, 32., .56 are example of real constants. 2.3e2 is also real constant which is represented in scientific notation. %f is used to display real values and %e for displaying real values in scientific notation. By default all the real values are treated as double precision values. If you want to specify a single precision real value, you can use the letter ‘f’ after the real value like -45.2f in the above program.

**/* C program to demonstrate character constants */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        clrscr();
        printf("%c \n", 'a');
        printf("%c \n", 'f');
        printf("%s \n", "hai");
        printf("%s \n", "hello");
        getch();
}
```

Explanation:
In the above program, ‘a’ and ‘f’ are single character constants, whereas “hai” and “hello” are string constants. %c is used to display single characters and %s is used to display strings.

**/* C program to demonstrate the use of variables */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a;          /*variable a declaration*/
        int b;          /*variable b declaration*/
        clrscr();
        a = 10;         /*Initialization of variable a*/
        b = 20;         /*Initialization of variable b*/
        printf("Value of a+b is: %d",(a+b));
        getch();
}
```

Explanation:
In the above program, a and b are variables which can hold integer values as specified by the **int** keyword in the declaration. Later, a is initialized to 10 and b is initialized to 20.

**/* C program to demonstrate different data types */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i = 10;
        unsigned int ui = 65535;
        short int si = 30;
        unsigned short int usi = 255;
        long int li = 2147483647;
        unsigned long int uli = 4294967295;
        float f = 23.4897897897;
        double d = 23.4235423542;
        long double ldouble = 24.562343546546;
        char ch = 'a';
        unsigned char uch = 'g';
        clrscr();
        printf("i = %d\n",i);
        printf("ui = %u\n",ui);
        printf("si = %hd\n",si);
        printf("usi = %hu\n",usi);
        printf("li = %ld\n",li);
        printf("uli = %lu\n",uli);
        printf("f = %f\n",f);
        printf("d = %f\n",d);
        printf("ldouble = %.Lf\n",ldouble);
        printf("ch = %c\n",ch);
        printf("uch = %c\n",uch);
        getch();
}
```

**/* C program to demonstrate user defined data types */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        /*Declaring sum as a user defined data type*/
        typedef int sum;
        /*Declaring days as a user defined data type*/
        enum days {mon,tue,wed,thu,fri,sat,sun};
        sum s;
        enum days day1;
        int a=10,b=20;
        clrscr();
        s = a+b;
        day1 = wed;
        printf("sum of a and b is: %d\n",s);
        printf("day1 value is: %d",day1);
        getch();
}
```

**/* C program to demonstrate const keyword */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        /*Declaring a constant pi*/
        const float pi = 3.142;
        float area;
        clrscr();
        area = pi*10*10;
        printf("Area of circle is: %f",area);
        getch();
}
```

**/* C program to demonstrate symbolic constants by using #define directive */**

```
#include<stdio.h>
#include<conio.h>
/*Declaring a symbolic constant PI*/
#define PI 3.142
main()
{
        float area;
        clrscr();
        area = PI*10*10;
        printf("Area of the circle is: %f",area);
        getch();
}
```

**/*C program to demonstrate use of scanf() function */**

```
#include<stdio.h>
#include<conio.h>
main()
{
        int  num1,num2;
        clrscr();
        printf("Enter the value of num1:\n");
        /* To accept the value into num1 */
        scanf("%d",&num1);
        /* To accept the value into num2 */
        printf("Enter the value of num2:\n");
        scanf("%d",&num2);
        printf("Sum of num1 and num2 is: %d",(num1+num2));
        getch();
}
```

# Operators

An operator is a symbol that tells a computer to perform certain mathematical or logical operations. Operators are used in programs to manipulate data and variables. Operators are usually a part of the mathematical or logical expressions. Generally the usage of an operator is as shown below:

*Operand1* Op *Operand2*

In the above format, operand1 and operand2 can be either data or variables or expressions. Op is the operator. In C, based on the number of operands on which an operator can operate, the operators are divided into three types namely: unary, binary and ternary. Unary operators work on single operand, binary operators work on two operands and ternary operators work on three operands. In C, based on the functionality, operators are classified into 8 categories. They are:

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

## Arithmetic Operators

C provides all the basic arithmetic operators as shown below. The arithmetic operators can operate on any built-in data type in C. The unary minus operator multiplies its single operand with -1. Integer division truncates any fractional part. The modulo division operator (%) produces the remainder of an integer division. The modulo operator cannot be used on floating point values.

| Operator | Meaning | Example |
|---|---|---|
| + | Addition or unary plus | a+b |
| - | Subtraction or unary minus | a-b |
| * | Multiplication | a*b |
| / | Division (Quotient) | a/b |
| % | Modulo (Remainder) | a%b |

## Relational Operators

In C, whenever there is a need to compare two values and make a decision based on the outcome of the comparison, we use relational operators. For example, we can compare a person's age with a value and based on whether the person's age is less than or equal or greater than the value, we may take a decision. The relational operators in C are as shown below:

| Operator | Meaning | Example |
|---|---|---|
| < | Less than | a<b |
| <= | Less than or equal to | a<=b |
| > | Greater than | a>b |
| >= | Greater than or equal to | a>=b |
| == | Equal to | a==b |
| != | Not equal to | a!=b |

The relational operators are generally used in decision making statements like if, else if and in looping statements like for, while, do while etc. Relational operators always evaluates to 0 (false) or 1 (true).

## Logical Operators

The relational operators are used to compare at most two values i.e. testing one condition. To test more than one condition, we use logical operators along with relational operators. The logical operators always evaluates to either 0 or 1 like relational operators. The logical operators available in C and their corresponding truth tables are as shown below:

| Operator | Meaning | Example |
|---|---|---|
| && | Logical AND | (a>b)&&(a>c) |
| \|\| | Logical OR | (a>b)\|\|(a>c) |
| ! | Logical NOT | !(a>b) |

Logical AND

| A | B | A&&B |
|---|---|---|
| F | F | F |
| F | T | F |
| T | F | F |
| T | T | T |

Logical AND

| A | B | A\|\|B |
|---|---|---|
| F | F | F |
| F | T | T |
| T | F | T |
| T | T | T |

Logical NOT

| A | !A |
|---|---|
| F | T |
| T | F |

## Assignment Operators

The assignment operators are used to assign value of an expression to a variable. The general assignment operator is = (equal). In C, there are some special assignment operators known as shorthand operators. The syntax of shorthand operators is as shown below:

```
Var op= exp;
```

In the above shown syntax, **var** is a variable, **op** is an arithmetic operator and **exp** can be any expression or value or a variable. The use of shorthand operators has three advantages:

1. Easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

The assignment operators in C are as shown below:

| Operator | Example |
|---|---|
| = | a = 10 |
| += | a+=10 (a=a+10) |
| -= | a-=10 (a=a-10) |
| *= | a*=10 (a=a*10) |
| /= | a/=10 (a=a/10) |
| %= | a%=10 (a=a%10) |

## Increment and Decrement Operators

The increment and decrement operators provided by C are used to increment or decrement the operand by a value of one. Both the increment and decrement operators are unary operators. These operators are used extensively inside for loop and while loop. There are two variations in how the increment and decrement operators can be used. They are as shown below:

```
Var++          Var--
      OR
++Var          --Var
```

In the above shown syntax, var++ is known as post increment and ++var is known as pre increment. Although both of them increment the variable by a value of one, they behave differently when they are used in expressions. In expressions, when post increment is applied, the value of the variable is used in the evaluation of the expression and after the expression is evaluated, the value of the variable is incremented by a value of one. When pre increment is applied, the value of the variable is incremented by one first and then that value is used for evaluation of the expression.
The increment and decrement operators available in C are:

| Operator | Meaning | Example |
|---|---|---|
| ++ | Increment by 1 | A++ or ++A |
| -- | Decrement by 1 | A-- or --A |

## Conditional Operator

The conditional operator "? :" in C, is a ternary operator, which operates on three operands. This operator is used to construct conditional expressions of the form:

```
exp1?exp2:exp3
```

In the above syntax, exp1, exp2 and exp3 refer to expressions. The "? :" works as follows: It evaluates the **exp1** first and then based on the result of the **exp1** it evaluates either **exp2** or **exp3**. If the result of **exp1** is true or non-zero, then **exp2** is evaluated or if the result of **exp1** is false or zero, then **exp3** is evaluated.

## Bitwise Operators

C supports a set of operators which operate at bit-level. These operators are known as bitwise operators. The bitwise operators are used for testing a bit, setting a bit, complementing a bit or for shifting the bits to left or right. The bitwise operators available in C are as shown below:

| Operator | Meaning | Example |
|---|---|---|
| & | Bitwise AND | a&b |
| \| | Bitwise OR | a\|b |
| ^ | Bitwise Exclusive OR | a^b |
| ~ | Bitwise NOT | ~a |
| << | Left Shift | a<<1 |
| >> | Right Shift | a>>1 |

The truth table for the bitwise operators is as shown below:

| 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 AND (&)*

The bitwise AND operator is used to test whether a bit at particular position is zero or one. If it is one, it returns a non-zero value.

```
20    =    00010100
      &
08    =    00001000

R     =    00000000
```

In the above example, we are testing whether the bit at position 4 in the number 20 is set to one or not. Since it is zero, the result is zero.

### *Bitwise OR (|)*

The bitwise OR operator is used to set a bit at particular position to one. Example usage of the bitwise OR operator is shown below:

```
20    =    00010100
      |
08    =    00001000

R     =    00011100
```

As you can see in the above example, we are trying to set the bit at position 4 in the number 20 to one. If the bit at the specified position is zero, the bitwise OR sets it to one. If it is already one, it leaves it as it is.

### *Bitwise Exclusive OR (^)*

The bitwise Exclusive OR returns zero, if both the bits are equal. Otherwise, it returns one. Example usage of this operator can be seen below:

```
20    =    00010100
      ^
08    =    00001000

R     =    00011100
```

### *Bitwise NOT (~)*

The bitwise NOT operator is used to complement every bit of the operand. If the bit is zero, it is set to one and if it is one, the bit is set to zero.

```
20    =    00010100
      ~
R     =    11101011
```

### *Left Shift (<<)*

The left shift operator is used to shift the bits in the operand to the left. The result of the left shift operation is equal to the operand multiplied by powers of two.

```
20      =       0 0 0 1 0 1 0 0
            <<
R       =       0 0 1 0 1 0 0 0
```

As seen in the above example, by left shifting the number 20 one time, we get the result as 40 which is 20 multiplied by 2. If we again left shift 20, we get the result 80 which is 20 multiplied by 4 and so on.

### *Right Shift (>>)*

The right shift operator is used to shift the bits in the operand to the right. The result of the right shift operation is equal to the operand divided by powers of two.

```
20      =       0 0 0 1 0 1 0 0
            >>
R       =       0 0 0 0 1 0 1 0
```

As seen in the above example, by right shifting the number 20 one time, we get the result as 10, which is actually dividing 20 by 2. If we again left shift 20, we get the result as 5 which is 20 divided by 4 and so on.

## Special Operators

C supports some special operators such as comma "," operator, sizeof operator, address "&" operator, pointer operator "*" and some others.

### *Comma Operator*

The **comma** "," operator is used to combine multiple related expressions together. A comma separated list of expressions is evaluated from left to right and the value of the right most expression is the value of the combined expression.

### *sizeof Operator*

The **sizeof** operator computes the size of an expression or variable or constant or a data type. This operator is used extensively for determining the length of an array, structure or union when the programmer does not know their actual sizes. The sizeof operator is also used in dynamic memory allocation. The general syntax of **sizeof** operator is as shown below:

```
sizeof(operand)
```

The operand can be either a value or variable or data type or an expression.

```
/* C program to demonstrate arithmetic operators */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1, num2;
        int result;
        clrscr();
        printf("Enter the value of num1:");
        scanf("%d",&num1);
        printf("Enter the value of num2:");
        scanf("%d",&num2);
        result = num1 + num2;
        printf("Value of adding num1 and num2 is: %d\n",result);
        result = num1 - num2;
        printf("Value of subtracting num1 and num2 is: %d\n",result);
        result = num1 * num2;
        printf("Value of multiplying num1 and num2 is: %d\n",result);
        result = num1 / num2;
        printf("Value of dividing num1 and num2 is: %d\n",result);
        result = num1 % num2;
        printf("Value of modulus on num1 and num2 is: %d",result);
        getch();
}
```

Output:

```
Enter the value of num1:20
Enter the value of num2:10

Value of adding num1 and num2 is: 30
Value of subtracting num1 and num2 is: 10
Value of multiplying num1 and num2 is: 200
Value of dividing num1 and num2 is: 2
Value of modulus on num1 and num2 is: 0
```

```
/* C program to demonstrate relational operators */

#include<stdio.h>
#include<conio.h>
main()
{
        int a, b;
        clrscr();
        printf("Enter the value of a:");
        scanf("%d",&a);
        printf("Enter the value of b:");
        scanf("%d",&b);
        if(a>b)
                printf("a is greater than b");
        else if(a<b)
                printf("a is less than b");
        else
                printf("a is equal to b");
        getch();
}
```

Output:

```
Enter the value of a:30
Enter the value of b:5

a is greater than b
```

```
/*C program to demonstrate logical operators */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1, num2;
        clrscr();
        num1 = 10, num2 = 20;
        if(num1<20 && num2<30)
                printf("Logical AND works!\n");
        if(num1<20 || num2<20)
                printf("Logical OR works!\n");
        if(!(num1==20))
                printf("Logical NOT works!");
        getch();
}
```

Output:
Logical AND works!
Logical OR works!
Logical NOT works!

```
/* C program to demonstrate assignment operators */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1;
        clrscr();
        num1 = 10;
        printf("num1 = %d\n",num1);
        num1 += 10;
        printf("num1 = %d\n",num1);
        num1 -= 10;
        printf("num1 = %d\n",num1);
        num1 *= 10;
        printf("num1 = %d\n",num1);
        num1 /= 10;
        printf("num1 = %d\n",num1);
        num1 %= 10;
        printf("num1 = %d\n",num1);
        getch();
}
```

Output:

```
num1 = 10
num1 = 20
num1 = 10
num1 = 100
num1 = 10
num1 = 0
```

```
/* C program to demonstrate increment and decrement */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1;
        clrscr();
        num1 = 10;
        printf("num1 = %d\n",num1++);
        printf("num1 = %d\n",++num1);
        printf("num1 = %d\n",--num1);
        printf("num1 = %d\n",num1--);
        getch();
}
```

Output:

```
num1 = 10
num1 = 12
num1 = 11
num1 = 11
```

```
/* C program to demonstrate conditional operator */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1;
        clrscr();
        num1 = 15;
        (num1>10)?(num1=20):(num1=5);
        printf("num1 = %d",num1);
        getch();
}
```

Output:
num1 = 20

```
/* C program to demonstrate bitwise operators */

#include<stdio.h>
#include<conio.h>
main()
{
        int num1, num2;
        clrscr();
        printf("Enter value of num1:");
        scanf("%d",&num1);
        num2 = 8;
        printf("Bitwise AND on num1 is: %d\n",(num1&num2));
        printf("Bitwise OR on num1 is: %d\n",(num1|num2));
        printf("Bitwise Ex-OR on num1 is: %d\n",(num1^num2));
        printf("Left shift on num1 is: %d\n",(num1<<1));
        printf("Right shift on num2 is: %d\n",(num1>>1));
        getch();
}
```

Output:

```
Enter value of num1:20

Bitwise AND on num1 is: 0
Bitwise OR on num1 is: 28
Bitwise Ex-OR on num1 is: 28
Left shift on num1 is: 40
Right shift on num2 is: 10
```

/* C program to demonstrate the use of **comma** operator and **sizeof** operator */

```
#include<stdio.h>
#include<conio.h>
main()
{
        int num1, num2;
        int sum;
        clrscr();
        /*Using the comma operator*/
        sum = (num1=5, num2=5, num1+num2);
        printf("Sum = %d\n",sum);
        /*Using the sizeof operator*/
        printf("Size of int is: %d bytes",sizeof(int));
        getch();
}
```

Output:
Sum = 10
Size of int is: 4 bytes

# Expression

An expression is a sequence of operands and operators that reduces to a single value. For example, the expression, 10+5 reduces to the value of 15. Based on the operators and operators used in the expression, they are divided into several types. Some of them are:

1. Integer expressions – expressions which contains integers and operators
2. Real expressions – expressions which contains floating point values and operators
3. Arithmetic expressions – expressions which contain operands and arithmetic operators
4. Mixed mode arithmetic expressions – expressions which contain both integer and real operands
5. Relational expressions – expressions which contain relational operators and operands
6. Logical expressions – expressions which contain logical operators and operands
7. Assignment expressions and so on… - expressions which contain assignment operators and operands

## Expression Evaluation

Expressions are evaluated using an assignment statement of the form:

```
variable = expression
```

In the above syntax, **variable** is any valid C variable name. When the statement like the above form is encountered, the expression is evaluated first and then the value is assigned to the variable on the left hand side. All variables used in the expression must be declared and assigned values before evaluation is attempted. Examples of expressions are:

```
x = a*b-c
y=b/c*a
z=a-b/c+d
```

Expressions are evaluated based on operator precedence and associativity rules when an expression contains more than one operator.

## Operator Precedence and Associativity

Every C operator has a precedence (priority) associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are different levels of operator precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators in the same level of precedence are evaluated from left to right or from right to left, based on the **associativity** property of an operator. In the below table we can look at the precedence levels of operators and also the associativity of the operators within the same level. Rank 0 indicates the lowest precedence and Rank 14 indicates highest precedence.

### Example 1

Let's understand the operator precedence and associativity rules with the help of an example. The expression that we consider for this example is:

**if(x==10+15&&y<10)**

In the above expression, the operators used are: ==,+,&& and <. By looking at the operator precedence table, + operator is at level 4, < operator is at level 6, == operator is at level 7 and && operator is at level 11. So clearly, the + operation is performed first. Then our expression becomes:

**if(x==25&&y<10)**

Now, since < operator has the next highest precedence, y<10 is evaluated. If we assume value of x is 20 and value of y is 5, then the value of y<10 is true. Then the == operator is evaluated. Since value of x is 20, x==25 evaluates to false. So, our expression becomes:

**if(false&&true)**

Now, the only operator left is &&. It is evaluated next and the result is false.

| | Operator | Associativity | Precedence |
|---|---|---|---|
| ()<br>[]<br>.<br>– > | Function call<br>Array subscript<br>Dot (Member of structure)<br>Arrow (Member of structure) | Left-to-Right | Highest 14 |
| !<br>~<br>–<br>++<br>––<br>&<br>*<br>(type)<br>sizeof | Logical NOT<br>One's-complement<br>Unary minus (Negation)<br>Increment<br>Decrement<br>Address-of<br>Indirection<br>Cast<br>Sizeof | Right-to-Left | 13 |
| *<br>/<br>% | Multiplication<br>Division<br>Modulus (Remainder) | Left-to-Right | 12 |
| +<br>– | Addition<br>Subtraction | Left-to-Right | 11 |
| <<<br>>> | Left-shift<br>Right-shift | Left-to-Right | 10 |
| <<br><=<br>><br>>= | Less than<br>Less than or equal to<br>Greater than<br>Greater than or equal to | Left-to-Right | 8 |
| ==<br>!= | Equal to<br>Not equal to | Left-to-Right | 8 |
| & | Bitwise AND | Left-to-Right | 7 |
| ^ | Bitwise XOR | Left-to-Right | 6 |
| \| | Bitwise OR | Left-to-Right | 5 |
| && | Logical AND | Left-to-Right | 4 |
| \|\| | Logical OR | Left-to-Right | 3 |
| ? : | Conditional | Right-to-Left | 2 |
| =, +=<br>*=, etc. | Assignment operators | Right-to-Left | 1 |
| , | Comma | Left-to-Right | Lowest 0 |

## Example 2

Let us try to evaluate an arithmetic expression as shown below:

**x = a-b/3+c*2-1**

Let a = 9, b =12, and c=3. Then our expression becomes:

**x = 9-12/3+3*2-1**

From the above table, we can see that the * and / operators are having higher precedence than + and – operators. Also, the * and / operators are at the same level of precedence, so we have to apply the associativity rules. Since the associativity rule is left-to-right, we apply the / operator first and the expression evaluates to:

**x = 9-4+3*2-1**

Next, we apply the * operator and the expression becomes:

**x = 9-4+6-1**

Next, we apply the first – operator as the – and + operators are at the same level and the associativity rule is from left to right. The expression becomes:

**x = 5+6-1**

Now, we apply the + operator and the expression become:

**x = 11-1**

Finally, we apply the – operator and the result is:

**x = 10**

**/* C Program to demonstrate expressions and operator precedence */**

```
#include<stdio.h>
#include<conio.h>
main()
{
   int m1=60,m2=60,m3=60;
   int average;
   average = m1+m2+m3/3;
   printf("Average is: %d",average);
}
```

Ouput:
Average is: 140

Explanation:
In the above program average is computed with the help of the expression, m1+m2+m3/3. Although this is mathematically correct, it is programmatically incorrect. We can see that the expression is made up of + and / operators. As / operator is having higher precedence than the + operator, m3/3 is evaluated first which gives 20. Then this 20 is added to m1 and m2 which gives 140. So, while writing expressions with multiple operators you should be careful.

# Type Conversion and Casting

Converting a data type of an expression to another data type is known as type conversion. Here expression refers to either a variable or value or sub expression. There are two types of conversions. They are:

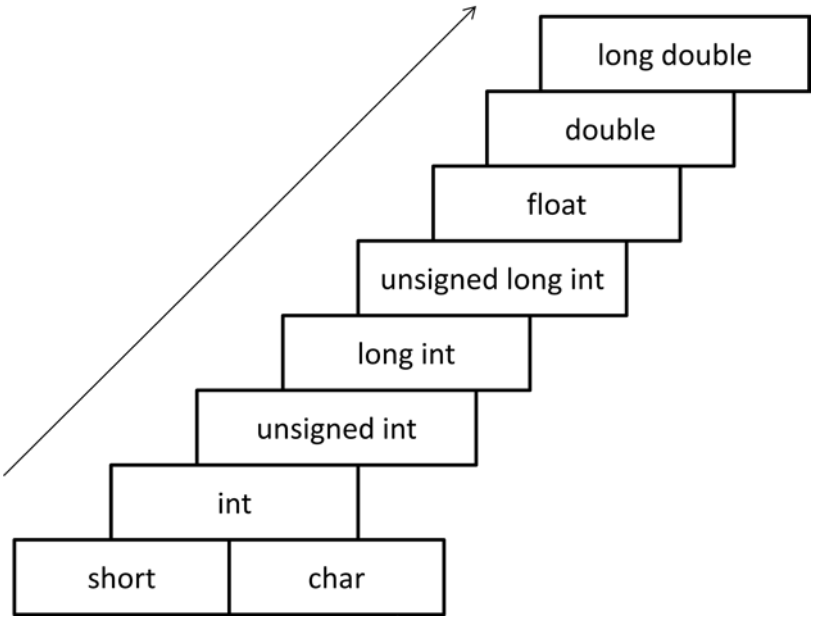
1. Implicit conversion or Coercion
2. Explicit conversion or Casting

## Implicit Conversion

While evaluating expressions, it is common to have operands of different data types. C automatically converts the data type of operands or any intermediate values to the proper type so that expression can be evaluated without losing any significance. This automatic conversion performed by C is known as **implicit conversion**.

In C, the expression on the right hand side of an assignment statement is evaluated first. So, C evaluates the expression based on the conversion hierarchy as shown in the below diagram. Finally, the type of the result acquired from the expression is again converted to match the type of variable on the left hand side. The conversion hierarchy is as shown below:

Conversion Hierarchy




### Example

Let's try to evaluate the following expression:

**x = l / i + i * f – d**

where, i and x are of the type **int**, f is of type **float**, d is of type **double** and l is of type **long int**. When the above expression is evaluated, the operand i in the sub expression l/i, is converted to long int since, l is a long int. In the sub expression i*f, i is converted into float, since f is of the type float. The result of the sub expression l/i is converted to float, as it will be added to the result of i*f which is already a float. Finally the result of the sub expression l/i+i*f is converted into double, as d is a double. After obtaining the final result which is of the type double, it is again converted into int as the variable x is of the type int.

## Explicit Conversion or Casting

In C programming, there will be instances where a programmer wants to change the type of an expression into another type which is not possible by the implicit conversion. In such cases, the programmer himself/herself specifies the type conversion. Such conversion is known as **explicit conversion or type casting**.

For example, let num1 and num2 are variables of integer type and value of num1 is 3 and value of num2 is 2. For the purpose of calculating the average of num1 and num2, let us declare another variable result of the type float. So the statement for calculating the average will be as shown below:

**result =( num1+num2)/2**

Mathematically the result of the above expression should be 2.5. But when we execute the above statement in a C program, we get the result 2.000000. Even though C applies implicit conversion from int to float, the result we obtained is wrong. In such case we apply explicit casting. The statement will be as follows:

**result = (float)(num1+num2)/2**

Now, the result of num1+num2 is converted into float. Since one operand is of type float, 2 will also be converted to float. The result of the whole expression on the right hand side is a float. Now, we obtain the correct value in the variable result which will be 2.500000.

The general syntax for explicit type casting is as shown below:

```
(type-name)expression
```

**/* C program to demonstrate implicit and explicit conversion */**

```
#include<stdio.h>
#include<conio.h>
main()
{
   int num1,num2;
   float average;
   num1=3,num2=2;
  /*Type casting from int to float*/
   average = (float)(num1+num2)/2;
   printf("Average is: %f",average);
}
```

Output:

```
Average is: 2.500000
```