CONTENT TABLE

FUNDAMENTAL OF COMPUTERS

Introduction to Computer:

1. **Charles Babbage** is the "Father" of the computer.
2. The word “computer” comes from the word “compute” which means to calculate. So a computer is normally considered to be a calculating device that can perform arithmetic operations.
3. Computer cannot do anything without a Program.
4. Computer is an advanced electronic device that takes raw data as input from the user and processes these data under the control of set of instructions (called program) and gives the result (output) and saves output for the future use. It can process both numerical and non-numerical (arithmetic and logical) calculations. The basic components of a modern digital computer are: Input Device, Output Device, and Central Processor. A Typical modern computer uses LSI Chips. Four Functions about computer are:

accepts data	Input
processes data	Processing
produces output	Output
stores results	Storage

- **Full form of Computer**
  - **C** : Common
  - **O** : Operating
  - **M** : Machine
  - **P** : Particularly
  - **U** : Used for
  - **T** : Trade
  - **E** : Education
  - **R** : Research

Generations of Computer:

First Generation (1945-1955)

1. In this generation **Vacuum tubes** were used to design the computer
2. Too bulky in size
3. Not portable
4. Limited commercial used
5. Large amount of heat generated
6. Air conditioning required
7. High Electricity Consumption
8. Very costly
9. Perform computations in **milliseconds**
10. Prone to Frequent hardware failures , so constant maintenance required
11. Unreliable

Ex: **UNIVAC I** (UNIVersal Automatic Computer).

- **Second Generation (1955 – 1964)**
  1. In this generation **Transistor** were used to design the computer
  2. Smaller in size as compared to first generation computer
  3. Better portability
  4. Wider commercial used but costly
  5. Less heat generated
  6. Air conditioning required
  7. Less prone to hardware failures
  8. Frequent maintenance required
  9. Perform computations in **microseconds** (able to reduce computational time from millisecond to microseconds)
  10. More reliable
- **Third Generation (1964 – 1975)**
  1. In this generation **IC(integrated circuit)** were used to design the computer
  2. Smaller in size as compared to previous generation computers
  3. Easily portable
  4. Widely used for various commercial applications all over the world
  5. Less heat generated as compared to previous generation computers
  6. maintenance cost is low because hardware failures are rare
  7. Perform computations in **nanoseconds** (able to reduce computational time from microseconds to nanoseconds)
  8. More reliable as compared to previous generation computers
  9. Less power required then previous generation computers
  10. Air conditioning required in many cases
- **Fourth Generation (1975 Onwards)**
  1. In this generation **microprocessors** were used to design the computer
  2. Uses LSI(large scale integration) and VLSI (very large scale integration) technique in microprocessor
  3. Small in size so easily portable
  4. Minimal labour and cost involved at assembly stage
  5. Very reliable
  6. Heat generated is negligible
  7. No Air conditioning required in most cases
  8. Much faster than previous generation computers
  9. Hardware failure is negligible so minimal maintenance required
  10. Cheapest among all generation

- **Fifth Generation (yet to come)**

  Scientists are now at work on fifth generation computers.

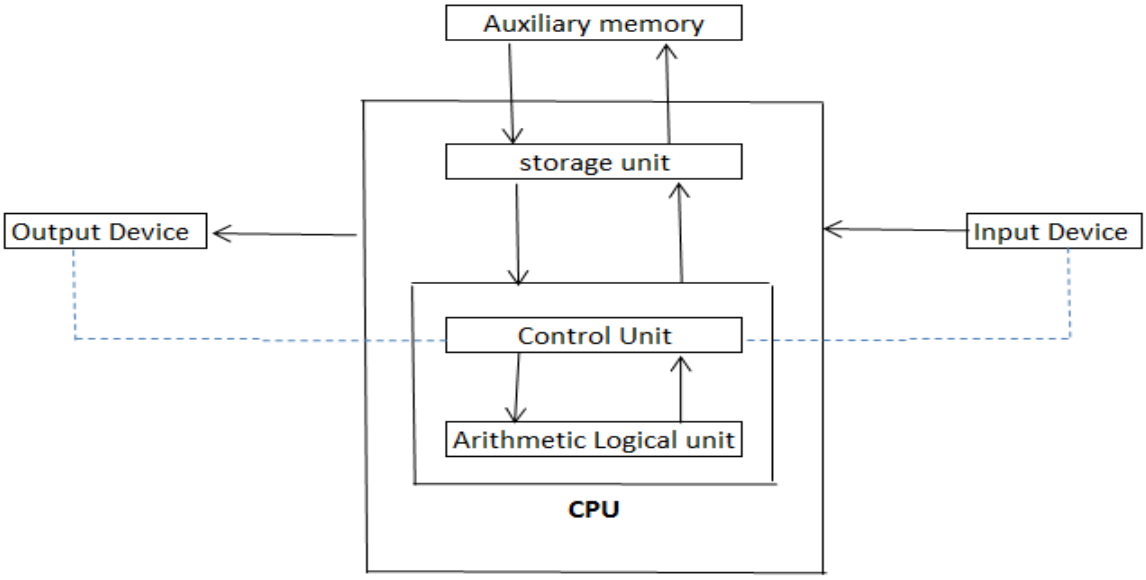
  They aim to bring us machines with genuine I.Q . The ability to reason logically and with real knowledge of the world.

  1. Used in parallel processing
  2. Used superconductors
  3. Used in speech recognition
  4. Used in intelligent robots
  5. Used in artificial intelligence

Characteristics of Computer:

1. **Speed: -** computer is a very fast machine. It can perform any task in few seconds.
2. **Accuracy: -** Computers are highly accurate machine and they never make mistakes. Errors can occur in a
   computer, but these are mainly due to human rather then technological weaknesses.
3. **Reliability: -** computers can be constructed and programmed in such a manner that they would always yield guaranteed results.
4. **Power of remembering: -** A computer can store and recall any amount of information because of its secondary storage (a type of detachable memory) capability. It saves the effort of inputs every time, which is useful for many computations in computer. The facility of the storage of data is on the temporary basis as well a long term basis.
5. **Versatility:-** Computer is useful everywhere in different respects such as in business, for personal use at home, in schools for educational purpose, in scientific and engineering application and so on. computer can be used for generic as well as specialized purpose.
6. **No feeling: -** computers are devoid of emotions. They have no feelings because they are machine.

Computer Organization:



Dig. Block Diagram of Digital Computer

- **Storage Units**

The storage units of a digital computer hold data and instructions that have been entered through the input units before and whilst they are being processed. They preserve the intermediate and final results before they are sent to the output devices. They also save the data for the later use. The various storage devices of a computer system are divided into two categories:

1. **Primary Memory/ Storage:** Random Access Memory. (RAM)

->This stores and delivers data very fast.
->RAM is generally used to hold the program being currently executed in the computer, the data being received from the input unit and the intermediate and final results of the program.
->RAM memory is temporary in nature: the data is lost when the computer is switched off.

- **Type of primary Storage:**

1.1. RAM:

- Stands for Random Access Memory.
- It stores temporary data into memory.
- When the power supply is switched off, the information stored inside the RAM is Lost.
- It stores essential data required to run programs, such as information and calculation.
- It is also referred to as read/write memory because information can be read from a RAM chip and also be written into it.

1.2 ROM:

- Stands for Read Only Memory
- typically used in computers to permanently hold data
- The information from the memory can only be read and it is not possible to write fresh information into it
- It is supplied by the computer manufacturer and user can not modify the program stored inside the ROM.
- When the power supply is switched off, the information stored inside the RAM is Lost.
- Example of ROM is a commercial CD purchased from a store; the manufacturers do not want you to alter what is stored on the disk.

- **Secondary Storage.**

->It stores several programs, documents, data bases etc. The programs that you run on the computer are first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory.

->Some of the commonly used secondary memory devices are Hard disk, CD, DVD, etc.

- **Input Device**

Digital computers need to receive data and instruction in order to solve any problem so they need to input data and instructions. Input can be entered from a keyboard, a mouse pointing device, a USB stick and the various types of photo storage cards used by digital cameras.

All input peripheral devices perform the following functions:

- Accept the data and instructions from the outside world.
- Convert it to a form that the computer can understand.
- Supply that converted data to the computer system for further processing.

➢ Output Units

The output units of a computer provide the information and results of an arithmetical computation - or some other kind of data processing, such as a search - to the outside world. Commonly used output units are printers and visual display units (VDUs), also known simply as "screens". Output data can also be sent to USB sticks and other types of data storage cards, such as those used by digital cameras. Output can also be uploaded to the Internet via a communications device.

All output peripheral devices perform the following functions:

- Accept output data and instructions from the computer.
- Convert it to a form that the outside world can understand.
- Supply that converted data to the outside world.

➢ Functional Units of a Computer System

In order to carry out the operations mentioned in the previous section the computer allocates the task between its various functional units. The computer system is divided into three separate units for its operation.

- Arithmetic logical unit
- Control unit.
- Central processing unit.

✓ Arithmetic Logical Unit

- After you enter data through the input device it is stored in the primary storage unit. The actual processing of the data and instruction are performed by Arithmetic Logical Unit.
- The major operations performed by the ALU are addition, subtraction, multiplication, division, logic and comparison. Data is transferred to ALU from storage unit when required.
- After processing the output is returned back to storage unit for further processing or getting stored.

✓ Control Unit (CU)

- The next component of computer is the Control Unit, which acts like the supervisor seeing that things are done in proper fashion.
- Control Unit is responsible for coordinating various operations.
- The control unit determines the sequence in which computer programs and instructions are executed.
- Things like processing of programs stored in the main memory, interpretation of the instructions and issuing of signals for other units of the computer to execute them.

- It also acts as a switch board operator when several users access the computer simultaneously.

✓ **Central Processing Unit (CPU)**

- The ALU and the CU of a computer system are jointly known as the central processing unit.
- You may call CPU as the brain of any computer system.
- It is just like brain that takes all major decisions, makes all sorts of calculations and directs different parts of the computer functions by activating and controlling the operations.

The CPU is like brain performs the following functions:

- It performs all calculations.
- It takes all decisions.
- It controls all units of the computer.

➢ Cache Memory

- Cache (pronounced *cash*) memory is extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip.
- The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed.

➢ Secondary memory/Auxiliary Memory:

- Secondary memory (or secondary storage) is the slowest and cheapest form of memory. It cannot be processed directly by the CPU. It must first be copied into primary storage (also known as RAM).
- Secondary memory devices include magnetic disks like hard drives and floppy disks ; optical disks such as CDs and CDROMs ; and magnetic tapes, which were the first forms of secondary memory.

Input-Output (IO) Units:

In computer, input/output or I/O is the communication between computer and the outside world, possibly a human or another information processing system.

- **HARD DISK :**

A hard disk drive (sometimes abbreviated as Hard drive, HD, or HDD) is a non-volatile memory hardware device that permanently stores and retrieves information. There are many variations, but their sizes are generally 3.5" and 2.5" for desktop and laptop computers respectively. A hard drive consists of one or more platters to which data is written using a magnetic head, all inside of an air-sealed casing.

The amount of data a hard drive can store depends on the storage space of the hard drive. Older hard drives had a storage size of several hundred megabytes (MB) to several gigabytes (GB). Newer hard drives have a storage size of several hundred gigabytes to several terabytes (TB).

- **FLOPPY DISK :**

A floppy diskette was first created in 1967by IBM. A floppy disk, also called a floppy, diskette or just disk, is a type of disk storage composed of a disk of thin and flexible magnetic storage medium, sealed in a rectangular plastic carrier lined with fabric that removes dust particles. Floppy disks are read and written by a floppy disk drive (FDD).

- **PEN DRIVE :**

Alternatively referred to as a **USB**, **flash drive**, **data stick**, **pen drive**, **memory unit**, **key chain drive** and **thumb drive**, a pen drive is a portable storage device.

A pen drive is most often used to store and transfer files between computers. It is often the size of a human thumb (hence the name), and it connects to a computer via a USB port.

Flash drives are an easy way to transfer and store information, and they are available in sizes ranging from 16 GB to 1 TB. Unlike a standard hard drive, the flash drive has no movable parts, containing only an integrated circuit memory chip that is used to store data.

- **CD READ/WRITE :**

Abbreviated as CD, a compact disc is a flat, round, optical storage medium invented by James Russell. The first CD was created at a Philips factory in Germany on August 17, 1982.

A small plastic disc on which music or other digital information is stored in the form of a pattern of metal-coated pits from which it can be read using laser light reflected off the disc.

The standard CD is capable of holding 72 minutes of music or 650MB of data. 80 minute CDs are also commonly used to store data and are capable of containing 700 MB of data. The picture is an example of the bottom of a standard compact disc.

➢ SCANNER :

A scanner is an input device that scans documents such as photographs and pages of text. When a document is scanned, it is converted into a digital format. This creates an electronic version of the document that can be viewed and edited on a computer.

Most scanners are flat bed devices, which means they have a flat scanning surface. This is ideal for photographs, magazines, and various documents.

➢ PRINTER :

A printer is an external hardware output device responsible for taking electronic data stored on a computer or computing device and generating a hard copy of that data. Printers are one of the most commonly used peripherals on computers and are commonly used to print text and photos.

Major types of computer printers:

I. Dot Matrix Printer

Also known as a pin printer, Dot matrix printers were first introduced by Centronicsin 1970. Dot matrix printers use print heads to shoot ink or strike an ink ribbon to place hundreds to thousands of little dots to form text and images. Today, Dot matrix printers are rarely used or found because of the low quality print compared to ink jet printers and laser printers.

II. Ink Jet Printer

The most popular printer for home computer users that prints by spraying streams of quick-drying ink on paper. The ink is stored in disposable ink cartridges, often a separate cartridge is used for each of the major colors. These colors are usually Black, Red/Magenta, Green/Cyan, and Yellow (CMYK).

III. Laser Printer

First developed at Xerox PARC by Gary Starkweather and released in 1971, a laser printer is a printer that utilizes laser technology to print images on the paper. Laser printers are often used in corporate, school, and other environments that require print jobs to be completed quickly and in large quantities.

➢ KEYBOARD :

A computer keyboard is an input device used to enter characters and functions into the computer system by pressing buttons, or keys. It is the primary device used to enter text. A keyboard typically contains keys for individual letters, numbers and special characters, as well as keys for specific functions. A keyboard is connected to a computer system using a cable or a wireless connection.

Most keyboards have a very similar layout. The individual keys for letters, numbers and special characters are collectively called the character keys. The layout of these keys is derived from the original layout of keys on a typewriter. The most widely used layout in the English language is called QWERTY, named after the sequence of the first six letters from the top left.

Number System

A number can be represented with different base values. We are familiar with the numbers in the base 10 (known as **decimal** numbers), with digits taking values 0,1,2,…,8,9.
A computer uses a **Binary** number system which has a base 2 and digits can have only two values: 0 and 1.
A decimal number with a few digits can be expressed in binary form using a large number of digits. Thus the number 65 can be expressed in binary form as 1000001.
The binary form can be expressed more compactly by grouping 3 binary digits together to form an octal number.
An **Octal** number with base 8 makes use of the EIGHT digits 0,1,2,3,4,5,6 and 7.
A more compact representation is used by **Hexadecimal** representation which groups 4 binary digits together. It can make use of 16 digits, but since we have only 10 digits, the remaining 6 digits are made up of first 6 letters of the alphabet. Thus the hexadecimal base uses 0,1,2,….8,9,A,B,C,D,E,F as digits.

To summarize

- ✓ Decimal : base 10 (0-9)
- ✓ Binary : base 2 (0-1)
- ✓ Octal: base 8 (0-7)
- ✓ Hexadecimal : base 16 (0-9A-F)

Decimal, Binary, Octal, and Hex Numbers

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

➢ Conversion of binary to decimal (base 2 to base 10)

Each position of binary digit can be replaced by an equivalent power of 2 as shown below.

2^{n-1}	2^{n-2}	…..	…..	2^3	2^2	2^1	2^0

Thus to convert any binary number replace each binary digit (bit) with its power and add up.

Example: convert (1011)2 to its decimal equivalent

Represent the weight of each digit in the given number using the above table

2^{n-1}	2^{n-2}	…..	…..	2^3	2^2	2^1	2^0
				1	0	1	1

Now add up all the powers after multiplying by the digit values, 0 or 1

$(1011)_2$

$= 2^3 * 1 + 2^2 * 0 + 2^1 * 1 + 2^0 * 1$

$= 8 + 0 + 2 + 1$

$= (11)_{10}$

Example2: convert (1000100)2 to its decimal equivalent

$= 2^6 * 1 + 2^5 * 0 + 2^4 * 0 + 2^3 * 0 + 2^2 * 1 + 2^1 * 0 + 2^0 * 0$

$= 64 + 0 + 0 + 0 + 4 + 0 + 0$

$= (68)_{10}$

➢ Conversion of decimal to binary (base 10 to base 2)

Here we keep on dividing the number by 2 recursively till it reduces to zero. Then we print the remainders in reverse order.

Example: convert (68)10 to binary

68/2 = 34 remainder is 0

34/ 2 = 17 remainder is 0

17 / 2 = 8 remainder is 1

8 / 2 = 4 remainder is 0

4 / 2 = 2 remainder is 0

2 / 2 = 1 remainder is 0

1 / 2 = 0 remainder is 1

We stop here as the number has been reduced to zero and collect the remainders in reverse order.

Answer = 1 0 0 0 1 0 0

Note: the answer is read from bottom (MSB, most significant bit) to top (LSB least significant bit) as $(1000100)_2$.

➢ Conversion of binary fraction to decimal fraction

In a binary fraction, the position of each digit(bit) indicates its relative weight as was the case with the integer part, except the weights to in the reverse direction. Thus after the decimal point, the first digit (bit) has a weight of ½ , the next one has a weight of ¼ ,
followed by 1/8 and so on.

2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…..	…..	…..
	1	0	1	1	0	0	0

The decimal equivalent of this binary number 0.1011 can be worked out by considering
the weight of each bit. Thus in this case it turns out to be
(1/2) x 1 + (1/4) x 0 + (1/8) x 1 + (1/16) x 1.

➢ Conversion of decimal fraction to binary fraction

To convert a decimal fraction to its binary fraction, multiplication by 2 is carried out repetitively and the integer part of the result is saved and placed after the decimal point. The fractional part is taken and multiplied by 2. The process can be stopped any time after the desired accuracy has been achieved.
Example: convert (0.68)10 to binary fraction.
0.68 * 2 = 1.36 integer part is 1
Take the fractional part and continue the process
0.36 * 2 = 0.72 integer part is 0
0.72 * 2 = 1.44 integer part is 1
0.44 * 2 = 0.88 integer part is 0
The digits are placed in the order in which they are generated, and not in the reverse order. Let us say we need the accuracy up to 4 decimal places. Here is the result.

Answer = 0. 1 0 1 0…..

Example: convert ($70.68)_{10}$ to binary equivalent.
First convert 70 into its binary form which is 1000110. Then convert 0.68 into binary form upto 4 decimal places to get 0.1010. Now put the two parts together.

Answer = 1 0 0 0 1 1 0 . 1 0 1 0….

➢ Octal Number System

•Base or radix 8 number system.
•1 octal digit is equivalent to 3 bits.
•Octal numbers are 0 to7. (see the chart down below)
•Numbers are expressed as powers of 8. See this table

8^{n-1}	8^{n-2}	……	……	8^3	8^2	8^1	8^0
					6	3	2

➢ Conversion of octal to decimal (base 8 to base 10)

Example: convert $(632)_8$ to decimal
$= (6 * 8^2) + (3 * 8^1) + (2 * 8^0)$
$= (6 * 6^4) + (3 * 8) + (2 * 1)$
$= 384 + 24 + 2$
$= (410)_{10}$

➢ Conversion of decimal to octal (base 10 to base 8)

Example: convert (177)10
to octal equivalent
177 / 8 = 22 remainder is 1
22 / 8 = 2 remainder is 6
2 / 8 = 0 remainder is 2

Answer = 2 6 1
Note: the answer is read from bottom to top as $(261)_8$, the same as with the binary case.
Conversion of decimal fraction to octal fraction is carried out in the same manner as decimal to binary except that now the multiplication is carried out by 8.
Example: convert (0.523)10 to octal equivalent up to 3 decimal places
0.523 x 8 = 4.184 ,its integer part is 4
0.184 x 8 = 1.472, its integer part is 1
0.472 x 8 = 3.776 , its integer part is 3
So the answer is $(0.413..)_8$

- **Conversion of decimal to binary (using octal)**

When the numbers are large, conversion to binary would take a large number of division by 2. It can be simplified by first converting the number to octal and then converting each octal into its binary form:

Example: convert $(177)_{10}$ to its binary equivalent using octal form
Step 1: convert it to the octal form first as shown above
This yields $(2\ 6\ 1)_8$
Step 2: Now convert each octal code into its 3 bit binary form, thus 2 is replaced by 010, 6 is replaced by 110 and 1 is replaced by 001. The binary equivalent is $(010\ 110\ 001)_2$
Example: convert $(177.523)_{10}$
to its binary equivalent up to 6 decimal places using octal form.
Step 1: convert 177 to its octal form first, to get $(2\ 6\ 1)_8$ and then convert that to the binary form as shown above, which is $(010\ 110\ 001)_2$
Step 2: convert 0.523 to its octal form which is $(0.413..)_8$
Step 3: convert this into the binary form, digit by digit.
This yields (0.100 001011…)
Step 4: Now put it all together
$(010\ 110\ 001\ .\ 100\ 001\ 011\ldots)_2$

- **Conversion of binary to decimal (using octal)**

First convert the binary number into its octal form. Conversion of binary numbers to octal simply requires grouping bits in the binary number into groups of three bits
•Groups are formed beginning with the Least Significant Bit and progressing to the MSB. Start from right hand side and proceed to left. If the left most group contains only a single digit or a double digit, add zeroes to make it 3 digits.

•Thus
$11\ 100\ 111_2$
$= (011\ \ 100\ \ 111)_2$
$= \ \ (3\ \ \ 4\ \ \ 7)_8$
And
$1\ 100\ 010\ 101\ 010\ 010\ 001_2$
$= (001\ \ 100\ \ \ 010\ \ 101\ \ 010\ \ \ 010\ \ \ 001)_2$
$= (1425221)_8$
Now it can be converted into the decimal form.

➢ Hexadecimal Number System

- ✓ Base or radix 16 number system.
- ✓ 1 hex digit is equivalent to 4 bits.
- ✓ Numbers are 0,1,2…..8,9, A, B, C, D, E, F. **Here** B is 11, E is 14
- ✓ Numbers are expressed as powers of 16.
  $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$, …

➢ Conversion of hex to decimal (base 16 to base 10)

Example: convert $(F4C)_{16}$ to decimal
$= (F * 16^2) + (4 * 16^1) + (C * 16^0)$
$= (15 * 256) + (4 * 16) + (12 * 1)$

➢ Conversion of decimal to hex (base 10 to base 16)

Example: convert $(4768)_{10}$ to hex.
= 4768 / 16 = 298 remainder 0
= 298 / 16 = 18 remainder 10 (A)
= 18 / 16 = 1 remainder 2
= 1 / 16 = 0 remainder 1

Answer: 1 2 A 0
Note: the answer is read from bottom to top , same as with the binary case.
$= 3840 + 64 + 12 + 0$
$= (3916)_{10}$

➢ Conversion of binary to hex

- ✓ Conversion of binary numbers to hex simply requires grouping bits in the binary numbers into groups of four bits.
- ✓ Groups are formed beginning with the LSB and progressing to the MSB.

Ex: $1110\ 0111_2 = E7_{16}$
$1\ 1000\ 1010\ 1000\ 0111_2$
$= 0001\ 1000\ 1010\ 1000\ 0111_2$
$= (1\ \ 8\ \ A\ \ 8\ \ 7)_{16}$

➢ Solve the following Expressions : Self Practice

1. $(354)_{10} = (?)_2$
2. $(456)_{10} = (?)_8$
3. $(ABCDE)_{16} - (8EEEF)_{16} = (?)_{16}$
4. $(54677)_8 - (4777)_8 = (?)_8$
5. $(5674)_8 + (6547)_8 = (?)_8$
6. $(65)_{10} = (0101)_?$

PROGRAMMING LANGUAGE

C Programming Language

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc.

C is the result of a development process that started with an older language called BCPL. BCPL was developed by Martin Richards, and it influenced a language called B, which was invented by Ken Thompson. B led to the development of C in the 1970s.

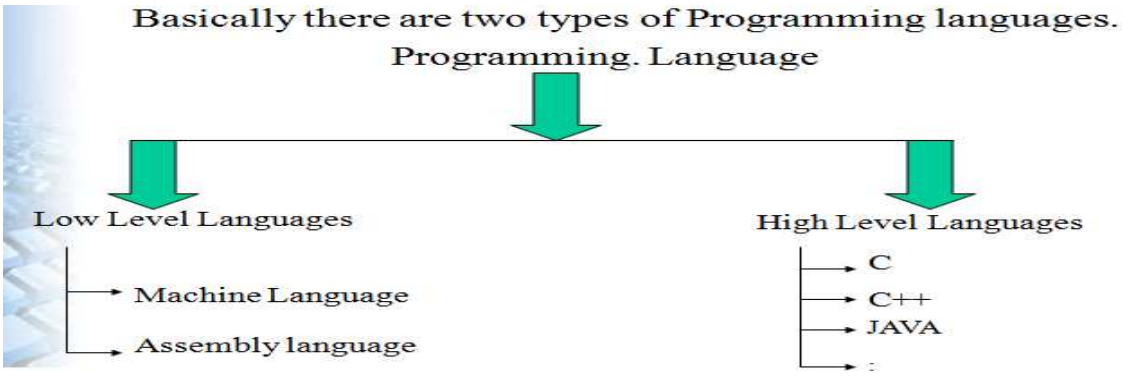
1) It is a case sensitive Language.
2) Extension of 'C' program file if ".C"
3) It is also called a Middle Level Language.

"C is a computer programming language".

What is Programming Language?

- Language is medium of communication.
- If two persons want to communicate with each other, they have to use a common language like Hindi, English, and Punjabi etc.
- In the same way when a user wants to communicate with a computer, the language that he/she uses for this purpose is known as programming language.
- Like BASIC, COBOL, C, C++, JAVA ……
- At the same time we know that computer understands only one language that is MACHINE LANGUAGE which is in 0 & 1 format.
- It is very difficult for us to learn this language.
- For Example two persons belonging to two different countries want to communicate then there may be a problem of common communication medium.
- To solve this problem there is a requirement of translator.

Types of Programming Language:



➢ Machine Language:

- It is in 0 and 1 format
- No need to translate.
- Programs are less understandable
- Difficult to write large programs
- Debugging is most difficult.
- Programs are machine dependent
- Not portable: portable to machine of the same architecture only

➢ Assembly Language:

- Uses mnemonic for programming (ADD,STORE)
- Need assembler to translate from assembly to machine language
- Programs are more understandable then machine language and less understandable then High level language.
- Difficult to write large programs but easy then machine language
- Debugging is easy then machine language and complex then high level language
- Programs are machine dependent
- Not portable: portable to processor of the same architectures only

➢ High Level Language:

- It uses natural language elements, which is easy to use.
- Need translator(compiler or assembler) to translate from high level language to object code.
- Easy to understand
- Easy to write large programs and suitable for software development
- Debugging is easier than others
- Programs are not machine dependent
- Programs are portable

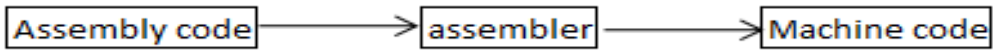
➢ Types of Translator:

There are mainly three types of translators,

1. Assembler
2. Complier
3. Interpreter

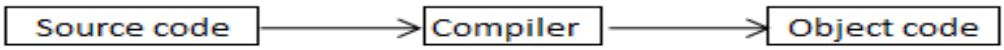
✓ Assembler:

- Assembly and machine language are low level language
- The assembly codes are easier to write but system can executes the machine code.
- So we need a translator (Assembler) to translate assembly code to machine code.



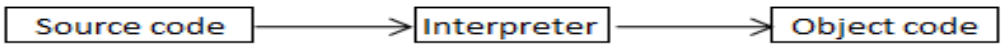
✓ Compiler:

- Used to translate high level source code to object code.
- Compiler scans all the lines of source program and list out all syntax errors at a time.
- It takes less time to execute.
- Object produced by compiler gets saved in a file. So file do not need to compile again and again



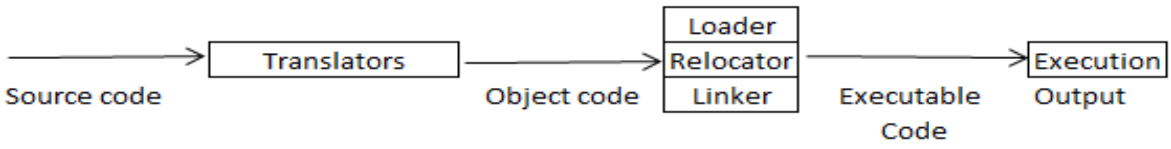
✓ Interpreter:

- Used to translate source code to object code.
- Interpreter scans one line at a time of source program if there is any syntax error , the execution of program terminates immediately.
- It takes more time to execute.
- Machine code produced by interpreter is not saved in any file . So we need to interpret the file each time(BASIC is an interpreter based language)



➢ Linker and loader:

The linker and loader is part of C translator(compiler and assembler)



✓ Loader:

- Loader is an operating system utility that copies program from a storage device to main memory, where they can be executed.
- It is a part of O.S. and used to load the program.
- Most loaders are transparent i.e. you can not directly execute them, but the O.S. use them when necessary.

✓ Linker:

- A linker is a program that combines object modules to form an executable program.
- Also called link editor
- It links the object modules with library functions and create executable program.

Generation of Language

First generation - machine language.

Second generation - assembly language.

Strong points:- what cannot be done in assembly language can just not be done at all with the given hardware.

Drawback:- control flow mechanisms (like if, while, for) are not directly available. I/O functions are not present, the program is not portable at all and programmer must have the knowledge of hardware

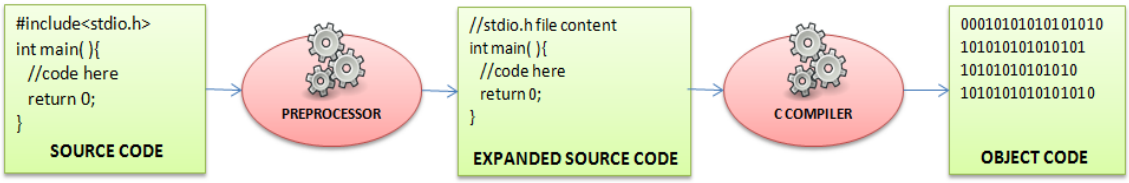
Third generation - was designed precisely to solve the problem of assembly .Languages like (COBOL, FORTRAN and Pascal providing high label I/O Facilities, hardware independence and so on. But these languages takes programmer away from hardware and they were not suitable for system label programming (task like writing OS etc).So there was a need of a language which should have feature of 3-GL with power of assembly. So the language **CPL (Combined Programming Language)** was came in picture .but CPL was very bulky and complex language so we had small derivative of CPL as **BCPL (Basic CPL)** by **Martin Richards**. BCPL further adapted by **Ken Thompson** to develop another language called **B**, and then **C** was developed by **Dennis Ritchie** based on B.

Compilation and Linking

Compilation process is a task of creating an executable code that is a multistage process divided into two components: **compilation** and **linking**. The total process of going from source code files to an executable might better be referred to as a **build**.

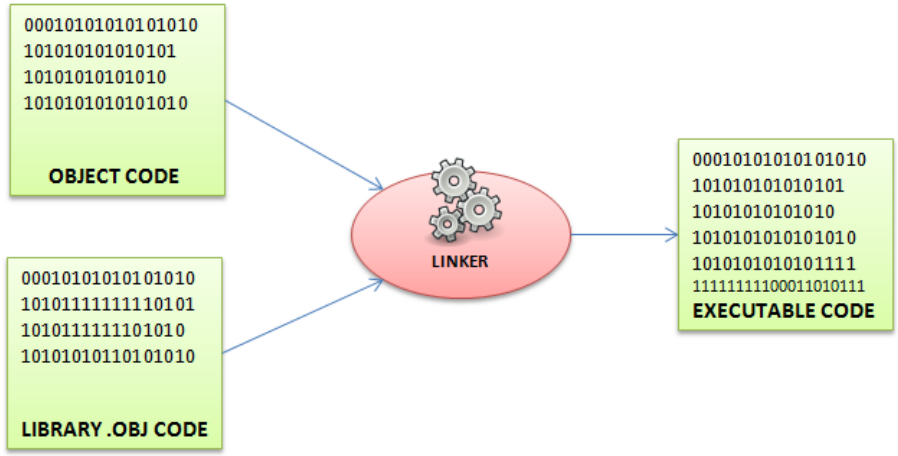
❖ Compilation

Compilation refers to the processing of source code files (.c) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. After compilation compiler creates object file the name <filename>.o or <filename>.obj (the extension will depend on compiler). Each of these files contains a translation of your source code file into a machine language file -- but you can't run them yet! You need to turn them into executables your operating system can use. That's where the linker comes in.



❖ Linking

Linking refers to the creation of a single executable file from multiple object files. In this step, the linker may complain about undefined functions. During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file(usually library functions) and will link later by linker.

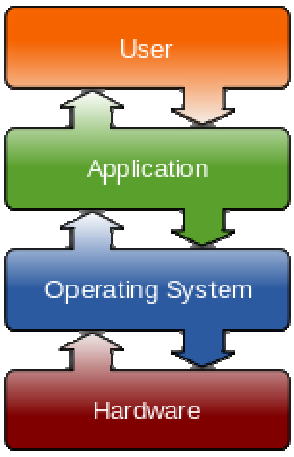


Operating System

An **operating system** (**OS**) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs usually require an operating system to function.

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently make a system call to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a computer—from cellular phones and video game consoles to supercomputers and web servers.



Examples of popular modern operating include Android, BSD, iOS, Linux, Mac OS X, Microsoft Windows, Windows Phone, and IBM z/OS. All these, except Windows and z/OS, share roots in UNIX.

➢ Types of operating system:

✓ Single Processing System:

- It has a single processor
- Only a single user can interact and runs a single program at a time.
- However the CPU is not utilized to its full potential, because it sits idle for most of the time. **Ex: MS DOS.**

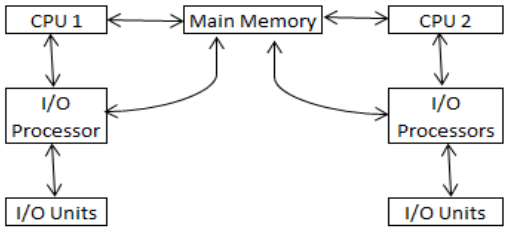
✓ Multiprogramming Operating System-

- Multiprogramming OS allows the execution of two or more different and independent programs by the same computer.
- It is used to increase the system utilization and efficiency.
- It reduces the CPU ideal time.

✓ Multiprocessing Operating System–

- Multiprocessing OS refers to a computer system's ability to support and utilizing more than one computer processor at a time.

- A multiprocessing operating system allows a program to run more than one central processing unit (CPU) at a time. **Ex: Linux , Windows 2000**



✓ Multitasking Operating System–

- An operating system that is capable of allowing multiple software processes to run at the same time.
- Multitasking is the ability to execute more than one task at the same time.
- In Multitasking only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all the programs at the same time. **Ex: Linux , Windows 2000**

✓ Multithreading Operating System–

- Multithreading Operating systems allows different parts of a single program to run concurrently.
- The ability of an OS to execute different parts of program, called threads, simultaneously without interfering with each other.
- **Ex: Linux , Windows 2000**

✓ Real-Time Operating System-

- There are many applications that require an immediate response from the computer.
- Real time means immediate response from the computer.
- Real-time operating systems are designed to allow computers to process and respond to input immediately.
- Used in stock market quotation, searching a criminal data file for a possible suspect may all be actions that need to be done without delay.

✓ Distributed System:

- A recent trend in computer system is to distribute computation among several processors. In contrast to tightly coupled system the processors do not share memory or a clock. Instead each processor has its own memory and clock.

- The processors communicate with one another through various communication lines such as high speed buses or telephone lines.
- These systems are usually referred to as distributed system or loosely coupled system.
- The processors in a distributed system may vary in size and function.
- They may include small micro processors, work stations, mini computers and large general purpose computer systems.
- These processors are referred to by a number of different names, such as sites, nodes, computers, and so on depending on the context in which they mentioned.
- There are variety of reasons for building distributed system, the major ones being:
    - Resource Sharing.
    - Computation Speedup.
    - Reliability and
    - Communication.

➢ Difference between Windows and Unix:

Windows	Unix
1. Windows is GUI based	1. Unix is text based
2. Windows is event driven	2 Unix is not event driven
3. It is multithreading OS	3 Multithreading and multi-tasking
4. Uses priority based preemptive scheduling.	4 Uses round robin preemptive scheduling
5. Windows is component based system.	5 Unix is integrated system.
6. Less secure but easy to understand.	6 More secure but difficult to understand.

➢ Difference between Windows and Linux

Windows	Linux
1. Developed and licensed by Microsoft.	1. Linux is open source meaning it is free under GNU.
2. Less secure	2 More secure
3. We can not change anything on Windows OS	3 It is user customizable allowing you to change code and add program
4. For printing more printer driver are available for windows	4 Less printer driver are available for Linux as it is not that popular.

ALGORITHM

1. The word algorithm comes from the name of a Persian mathematician, Abu Ja'far Mohammed ibn Musa al Khowarizmi (825 A.D)
2. An algorithm is an unambiguous set of direction specified a sequence of operation designed to solve a particular type of problem
3. Algorithm is progress to step wise to step wise problem solve to any problem
4. Algorithm may be formally defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired result will be obtained.
5. An algorithm is a list of steps for solving a problem.
6. It is a sequence of finite instructions, used for calculation and data processing.
7. It takes some value or set of values as input and produces some values as output.
8. It specifies how to compute the results for any given set of arguments.

- **Characteristics of an algorithm**
  1. **Input: -** valid inputs are clearly specified
  2. **Output: -** can be proved to produce the correct output given a valid input
  3. **Definiteness: -** Strictly and unambiguously(clearly) specified
  4. **Effectiveness: -** steps are sufficiently simple and basic.
  5. **Finiteness: -** terminates after a finite number of steps

1. ADDITION OF GIVEN TWO NUMBER.

```
STEP  1  :  START
STEP  2  :  A=10, B=20
STEP  3  :  C=A+B
STEP  4  :  PRINT "SUM" C
STEP  5  :  STOP
```

2. SIMPLE INTEREST OF GIVEN PRINCIPAL, RATE AND TIME.

```
STEP  1   :  START
STEP  2   :  INPUT P, R, T
STEP  3   :  SI=P*R*R/100
STEP  4   :  PRINT "SIMPLE INTERST" SI
STEP  5.  :  STOP
```

3. FIND OUT AREA AND PERIMETER OF GIVEN RADIUS OF CIRCLE

STEP	1	:	START	
STEP	2	:	INPUT R	
STEP	3	:	PERI=2*3.14*R	
STEP	4	:	AREA=3.14*R*R	
STEP	5	:	PRINT “PERIMETER” PERI	
STEP	6	:	PRINT “AREA”, AREA	
STEP	7	:	STOP	

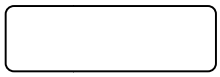
4. WRITE AN ALGORITM TO SWAPING OF GIVEN TWO NUMBER

STEP	1	:	START	
STEP	2	:	INPUT A, B	A=10 B=20
STEP	3	:	C=A	C=A=10
STEP	4	:	A=B	A=B=20
STEP	5	:	B=C	B=C=10
STEP	6	:	PRINT A, B	A=20 B=10
STEP	7	:	STOP	

FLOW CHART

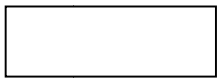
1. A flow chart is a graphical or symbolic representation of a process.
2. Each step in the process is represented by a different symbol and contains a short description of the process step.
3. The flow chart symbols are linked together with arrows showing the process flow direction.
4. Different flow chart symbols have different meanings. The most common flow chart symbols are:
5. In the case of complicated problems it is desirable to draw a flow chart. This process of showing diagrammatically the method of solution is referred to as flow chart. A flow chart indicated the direction of flow of a process, relevant operation and computations, points of decision and other information which is part of the solution

Terminator: An oval flow chart shape indicating the start or end of the process.



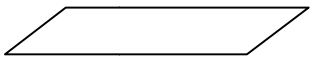
Start, Stop, Begin, End

Process: A rectangular flow chart shape indicating a normal process flow step.



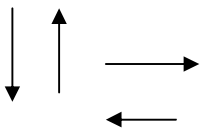
Processing Box

Data: A parallelogram that indicates data input or output (I/O) for a process.



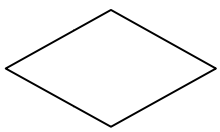
Input, Output

Direction : A Arroy key to indicate to flow of program.



Flow Direction

Decision : A diamond flow chart shape indication a branch in the process flow.

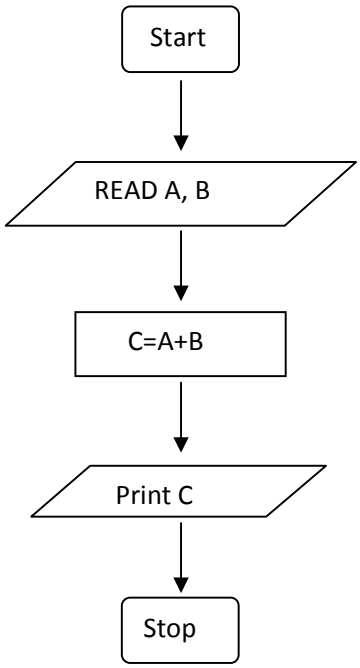


Decision Box

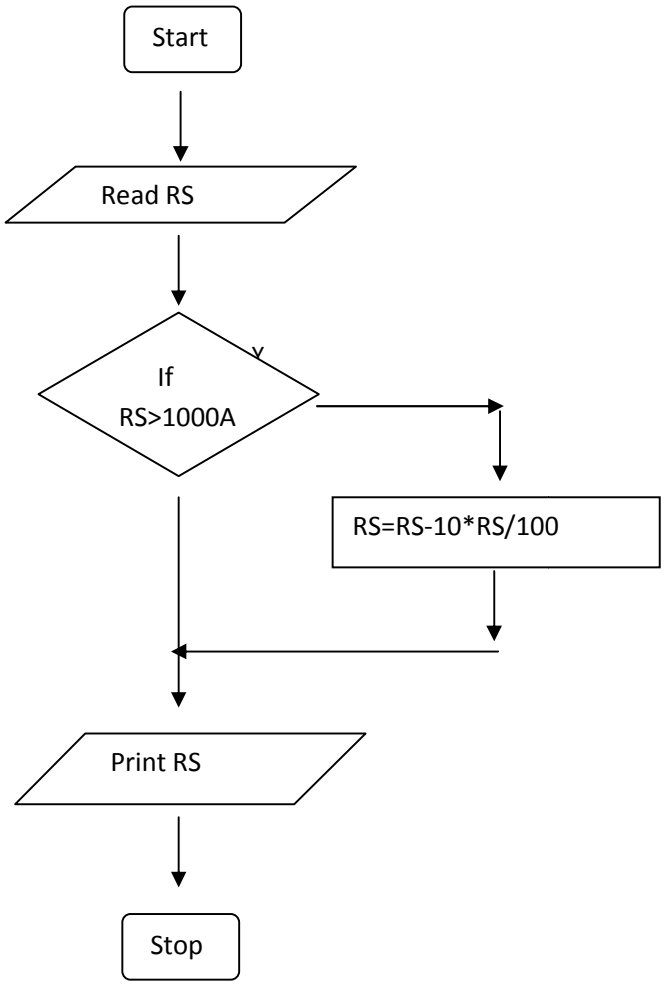
Connector: A small, labeled, circular flow chart shape used to indicate a jump in the process flow. (Shown as the circle with the letter “A”, below.)

Document: Used to indicate a document or report (see image in sample flow chart below).

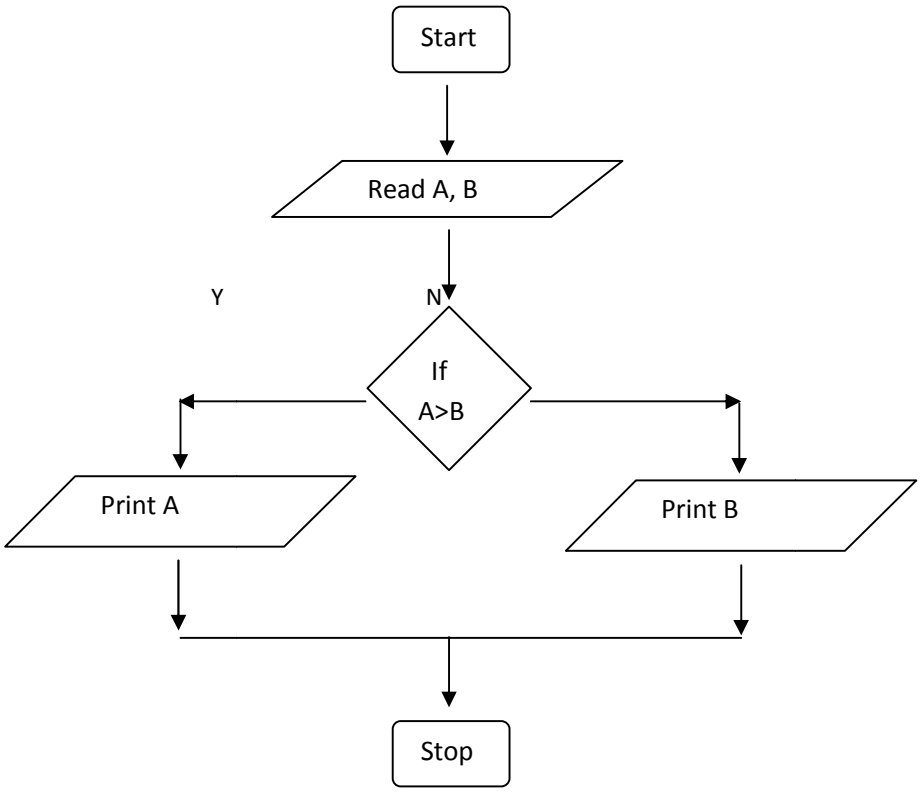
1 WRITE A FLOW CHART ACCEPT TWO NUMBER FIND OUT THE SUM



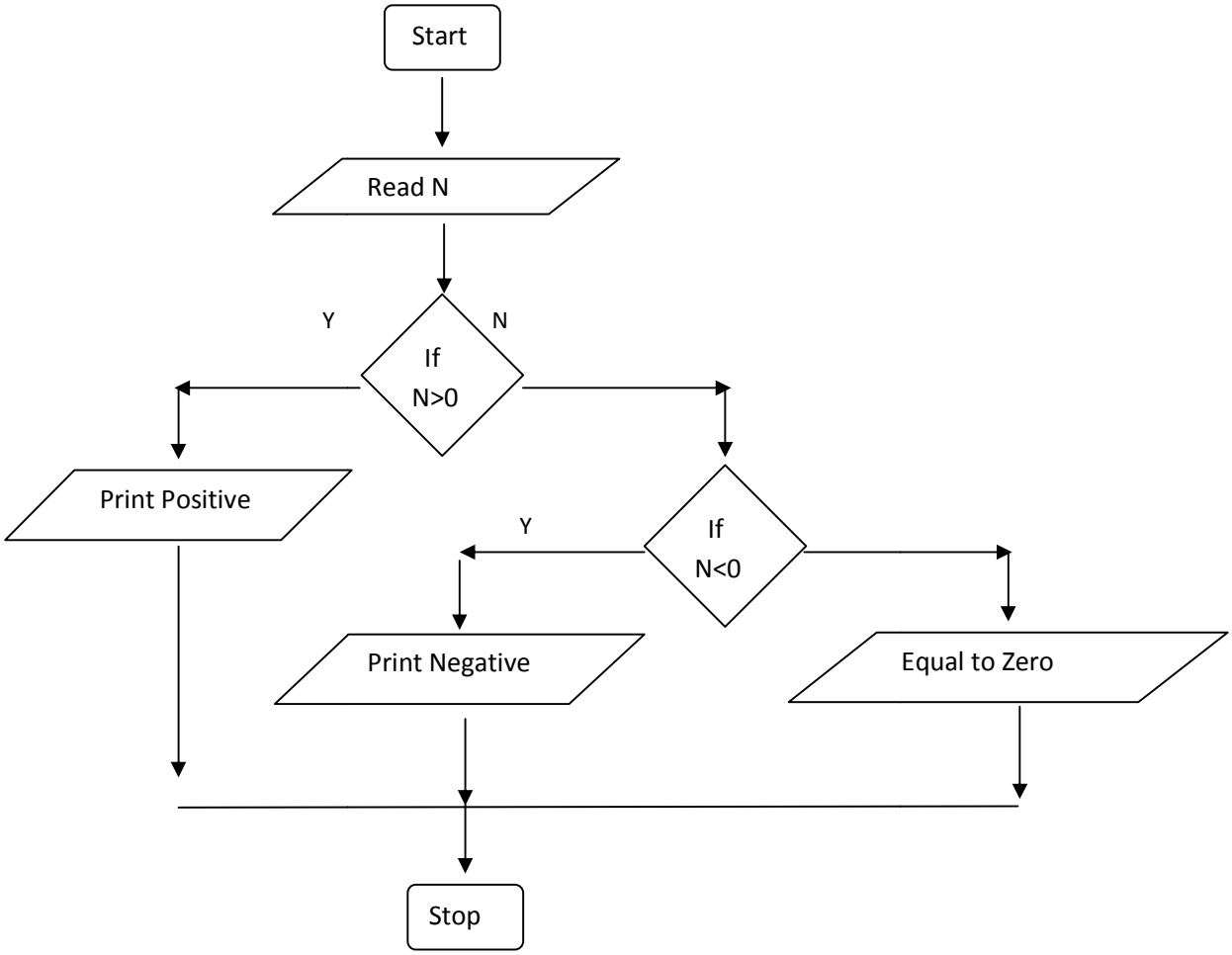
2 WRITE A FLOW CHART GIVEN A DISCOUNT 10% IF PURCHASING MORE THE RS. 1000. OTHER WISE NO ANY DISCOUNT.



3 WRITE A FLOW CHART ACCEPT TWO NUMBER FIND OUT THE MAXUMUM NUMBER



4 WRITE A FLOW CHART ACCPET ANY NUMBER FIND OUT THE GIVEN NUMBER IS "POSSITIVE NUMBER" , "NEGATIVE NUMBER" OR "EQUAL TO ZERO"



C LANGUAGE FUNDAMENTALS

Data Types Of C :

- C language is rich in its data type.
- Data type tells the type of data, that you are going to store in memory.
- It gives the information to compiler that how much memory (No. of bytes) will be stored by the data.

The C language supports a number of data types, all of which are necessary in writing programs. The basic data types of C language are defined by the ANSI standard are listed in Table.

Type	Size	Description
char	1byte	Used for characters or integer variables.
int	2 or 4 bytes	Used for integer values.
float	4 bytes	Single precision floating point values
double	8 bytes	Double precision floating point values

In addition to these data types, some of them may be used with a modifier that affects the characteristics of the data object. These modifiers are listed in Table.

Modifier	Description
long	Forces a type int to be 4 bytes (32 bits) long and forces a type double to be larger than a double (but the actual size is implementation defined). Cannot be used with short.
short	Forces a type int to be 2 bytes (16 bits) long. Cannot be used with long.
unsigned	Causes the compiler (and CPU) to treat the number as containing only positive values. Because a 16-bit signed integer can hold values between –32,768 and 32,767, an unsigned integer can hold values between 0 and 65,535. The unsigned modifier can be used with char, long, and short (integer) types.

The essence of all the data types that we have learnt so far has been captured in Table.

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

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Figure 1.2 shows the valid alphabets, numbers and special symbols allowed in C.

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

Figure 1.2

‘C’ Programming Tokens:

1. Keywords
2. Identifiers
3. Variables
4. Constants
5. Special Symbols
6. Operators

1. **KEYWORDS :**
   - Keywords are the reserved words whose meaning has already been explained to the C compiler.
   - C has 32 keywords.
   - These keywords combined with a formal syntax form a C programming language.
   - Rules to be followed for all programs written in C:
     - All keywords are in lower case.
     - C is case sensitive so **int** is different from **Int**.
     - The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer.
   - List of C programming keywords.

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

2. IDENTIFIERS:

- Identifiers refer to the naming of programming elements like 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 are permitted, although lowercase letters are commonly used.
- The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.
- Some legal identifier :

  arena, s_count
  marks40
  class_one
- Some illegal identifiers:

  1stsst
  oh!god
  start….end
- The number of characters in the variable that are recognized differs from compiler to compiler.
- An identifier cannot be the same as a C keyword.

3. VARIABLE :

- Variables are named locations in memory that are used to hold a value that may be modified by the program.
- Unlike constants that remain unchanged during the execution of a program.
- A variable may take different values at different times during execution.
- The syntax for declaring a variable is –

  DataType IdentifierName ;

  Example. int num;
  long int sum , a;

4. CONSTANT :

- Constants are named locations in memory that are used to hold a value that cannot be modified by the program.
- Constants remain unchanged during the execution of a program.
- **const keyword** is used to declare constant data.
- The syntax for declaring a constant is –

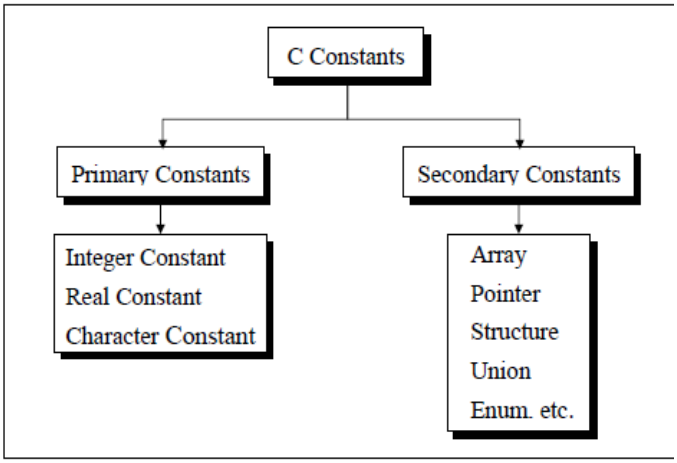
  const DataType IdentifierName ;

  Example. const float PI = 3.14f;

- **Types of C Constants**

C constants can be divided into two major categories:

- Primary Constants
- Secondary Constants



5. OPERATOR :

- Operator is a symbol that's operates on one or more operands and produces output.

  Exam - c = a + b ;

  In the above Example the symbols + and = are operators that operate on operands a, b , and c .

6. SPECIAL OPERATOR :

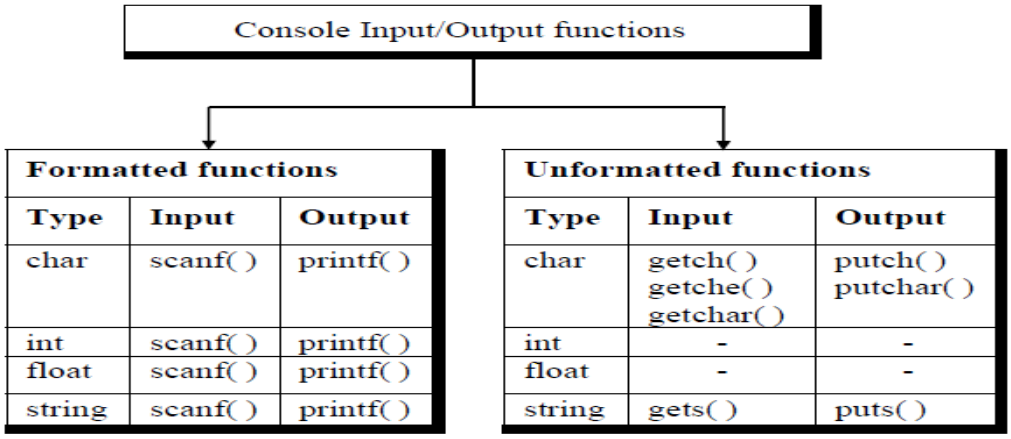
- ! , @, #, $, & , * , …..
- These all symbols are called Special Symbols.
- Every symbol has its special meaning in different respect at different place that's why it is called Special Symbols.

Standard Input/Output Assignments in C:

There are numerous library functions available for performing input and output.

❖ Console Input/output functions:

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions.



Formatted functions		
Type	**Input**	**Output**
char	scanf( )	printf( )
int	scanf( )	printf( )
float	scanf( )	printf( )
string	scanf( )	printf( )

Unformatted functions		
Type	**Input**	**Output**
char	getch( ) getche( ) getchar( )	putch( ) putchar( )
int	-	-
float	-	-
string	gets( )	puts( )

1. **printf() Function: printf** function is used to display output to the output screen.

The general syntax of this function is as follows

```
printf ( "format string", list of variables ) ;
```

The format string can contain:

a. Characters that are simply printed as they are
b. Conversion specifications that begin with a % sign
c. Escape sequence begin with (\) symbol

For example, look at the following program:

```
int main( )
{
    int avg = 346 ;
    float per = 69.2 ;
    printf ( "Average = %d\nPercentage = %f", avg, per ) ;
   return 0;
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

The **%d** and **%f** used in the **printf( )** are called format specifiers. They tell **printf( )** to print the value of **avg** as a decimal integer and the value of per as a float. Following is the list of format specifiers that can be used with the **printf( )** function.

Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

We can provide following optional specifiers in the format specifications.

% [flag] [width] [.precision] conversion

Now a short explanation about these optional format specifiers.

[width]

Specifying the field width can be useful in creating tables of numeric values.

The field-width specifier tells **printf( )** how many columns on screen should be used while printing a value. For example, **%10d** says, "print the variable as a decimal integer in a field of 10 columns". If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left.

[flag]

Flag include the minus sign in format specifier (as in **%-10d**), this means left justification is desired and the value will be padded with blanks on the right. Here is an example that should make this point clear.

```
int  main( )
{
     int weight = 63 ;
     printf ( "\nweight is %d kg", weight ) ;
     printf ( "\nweight is %2d kg", weight ) ;
     printf ( "\nweight is %4d kg", weight ) ;
     printf ( "\nweight is %6d kg", weight ) ;
```

```
    printf ( "\nweight is %-6d kg", weight ) ;
    return 0;
}
```

The output of the program would look like this ...

weight is 63 kg
weight is 63 kg
weight is 63 kg
weight is 63 kg
weight is 63 kg

[.precision]

Specifying the field width can be useful in creating tables of numeric values, as the following program demonstrates.

```
int main( )
{
    printf ( "\n%10.1f %10.1f %10.1f", 5.0, 13.5, 133.9 ) ;
    printf ( "\n%10.1f %10.1f %10.1f", 305.0, 1200.9, 3005.3 );
    return 0;
}
```

This results into a much better output...

5.0 13.5 133.9
305.0 1200.9 3005.3

❖ Escape Sequences:

Escape sequences are used in the C language. These are character combinations that comprise a backslash (\) followed by some character. They give results such as getting to the next line or a tab space. They are called *escape sequences* because the backslash causes an "escape" from the normal way characters are interpreted by the compiler.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

2. **scanf() Function :** Allows us to enter data from keyboard that will be formatted in a certain way. The general form of **scanf( )** statement is as follows:

scanf ("format string", list of addresses of variables) ;

For example:

```
int main( )
{
   int  a, b;
   printf(“Enter the value for a : ”);
   scanf(“%d”, &a);
   printf(“Enter the value for b : ”);
   scanf(“%d”, &b);
   printf(“User Input is : \n A = %d and B = %d”, a, b);
   return 0;
}
```

OUTPUT:

```
Enter the value for a : 10
Enter the value for b : 20
User Input is :
A = 10 and B = 20
```

Note that we are sending addresses of variables (addresses are obtained by using ‘**&**’ the ‘address of’ operator) to **scanf( )** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s).

❖ Unformatted Console I/O Functions:

There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters.

- ✓ **getch( )** just returns the character that you typed without echoing it on the screen.
- ✓ **getche( )** function 'e' means it echoes (displays) the character that you typed to the screen.
- ✓ **getchar( )** works similarly and echo's the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed.
- ✓ **gets( )** works similarly as getchar( ) but to accept string from user.

```
int main( )
{
      char ch ;
      printf ( "\nPress any key to continue" ) ;
      getch( ) ; /* will not echo the character */
      printf ( "\nType any character" ) ;
      ch = getche( ) ;            /* will echo the character typed */
      printf ( "\nType any character" ) ;
      getchar( ) ;                /* will echo character, must be followed by enter key */
      printf ( "\nContinue Y/N" ) ;
      fgetchar( ) ;               /* will echo character, must be followed by enter key */
      return 0;
}
```

- ✓ **putc(ch)** print / display the character on the screen.
- ✓ **putchar(ch)** print / display the character on the screen.
- ✓ **puts(str)** print / display the string on the screen, can print only one string at a time.

Operators in C :

An operator is a symbol that operates on a certain data type and produces the output as the result of the operation. In C, you can combine various operators of similar or different categories and perform an operation. In this case the C compiler tries to solve the expression as per the rules of precedence.

❖ **Category of operators**

Unary Operators : A unary operator is an operator, which operates on one operand.

Binary : A binary operator is an operator, which operates on two operands

Ternary : A ternary operator is an operator, which operates on three operands.

Following are the different types of Operators

(1) Arithmetic Operator

The arithmetic operator is a binary operator, which requires two operands to perform its operation of arithmetic. Following are the arithmetic operators that are available.

Operator	Description
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulo or remainder

(2) Relational Operators

Relational operators compare between two operands and return in terms of true or false i.e. 1 or 0. In C and many other languages a true value is denoted by the integer 1 and a false value is denoted by the integer 0.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Example:

Suppose we have three integer variables a, b and c with values 1,2 and 3 respectively.

Expression	Returns	Meaning
a<b	1	TRUE
(a+b)>=c	1	TRUE
(b+c)>(a+5)	0	FALSE
(c!=3)	0	FALSE
B==2	1	TRUE

(3) Logical Operators

A logical operator is used to compare or evaluate logical and relational expressions. There are three logical operators available in the C language.

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical NOT

Example

Let us take three variables a, b, and c and assume a = 5,b=10 and c=7

```
a>b && a+b<c
```

Here a>b will return false hence 0 and a+b>c will return true hence 1. Therefore, looking at the truth table 0 AND 1 will be 0. From this we can understand, that this expression will evaluate to false.

Let us look at the same example using Logical OR (||)

```
a&gt;b || a+b&gt;c
```

Here a>b will return false hence 0 and a+b>c will return true hence 1. Therefore, looking at the truth table 0 OR 1 will be 1. From this we can understand, that this expression will evaluate to true.

(4) Assignment Operator

An assignment operator (=) is used to assign a constant or a value of one variable to another.

Example:

```
a = 5;
b = a;
interestrate = 10.5
result = (a/b) * 100;
```

Important Note:

Remember that there is a remarkable difference between the equality operator (==) and the assignment operator (=). The equality operator is used to compare the two operands for equality (same value), whereas the assignment operator is used for the purposes of assignment.

Multiple assignments:

You can use the assignment for multiple assignments as follows:

```
a = b= c = 10;
```

At the end of this expression all variables a, b and c will have the value 10. Here the order of evaluation is from right to left. First 10 is assigned to c, hence the value of c now becomes

10. After that, the value of c (which is now 10) is assigned to b, hence the value of b becomes 10. Finally, the value of b (which is now 10) is assigned to a, hence the value of a becomes 10.

Arithmetic Assignment Operators

Arithmetic Assignment operators are a combination of arithmetic and the assignment operator. With this operator, the arithmetic operation happens first and then the result of the operation is assigned.

Example	Expands as
Sum+=3	Sum = sum + 3
Count-=4	Count -= 4
Factorial*=num	Factorial = factorial * num
Num/=10	Num = num /10
A%=3	A = a % 3

Operators

+=, -=, *=, /=, %=

(5) Increment and Decrement Operators

These operators also fall under the broad category or unary operators but are quite distinct than unary minus.

The increment and decrement operators are very useful in C language. They are extensively used in for and while loops.

The ++ operator increments the value of the variable by one, whereas the -- operator decrements the value of the variable by one.

These operators can be used in either the postfix or prefix notation as follows:

- ✓ **Postfix notation**

In the postfix notation, the value of the operand is used first and then the operation of increment or decrement takes place, e.g. consider the statements below:

```
int a,b;
a = 10;
b = a++;
printf("%d\n"  ,a);
printf("%d\n"  ,b);
```

OUTPUT :

```
11
10
```

- ✓ **Prefix notation**

In the prefix notation, the operation of increment or decrement takes place first after which the new value of the variable is used.

```
int a,b;
a = 10;
b = ++a;
printf("%d\n"  ,a);
printf("%d\n"  ,b);
```

OUTPUT :

```
11
11
```

(6) Bitwise Operators :

One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various Bitwise Operators available in C are shown in table

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
>>, <<	Right and Left Shift

TRUTH TABLE FOR BITWISE OPERATION					
BITS		AND	OR	XOR	NOT
A	B	A&B	A|B	A^B	~A
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

✓ Right Shift Operator

The right shift operator is represented by >>. It needs two operands. It shifts each bit in its left operand to the right. The number of places the bits are shifted depends on the number following the operator (i.e. its right operand).

Thus, **ch >> 3** would shift all bits in **ch** three places to the right. Similarly, **ch >> 5** would shift all bits 5 places to the right.

For example, if the variable **ch** contains the bit pattern 11010111, then, **ch >> 1** would give 01101011 and **ch >> 2** would give 00110101.

6>>2 = 24

✓ Left Shift Operator

This is similar to the right shift operator, the only difference being that the bits are shifted to the left, and for each bit shifted, a 0 is added to the right of the number.

6<<2 = 1

(7) Conditional Operator

A conditional operator checks for an expression, which returns either a true or a false value. If the condition evaluated is true, it returns the value of the true section of the operator, otherwise it returns the value of the false section of the operator.

Its general structure is as follows:

```
Expression1 ? expression 2 (True Section): expression3 (False Section)
```

Example:

```
a=3,b=5,c;
 c = (a<b) ? a+b : b-a;
```

The variable c will have the value 2, because when the expression (a>b) is checked, it is evaluated as false. Now because the evaluation is false, the expression b-a is executed and the result is returned to c using the assignment operator.

❖ Type Conversion

When an operators and operands are of different data types, C automatically converts them to the same data type. This can affect the results of mathematical expressions. This is called type conversion. In C type conversion can be done by two ways, viz.

Automatic Conversion (Arithmetic Promotion)
When a value is converted to a higher type, it is said to be promoted. Lets a look at the specific rules that govern the evaluation of mathematical expressions.

Rule 1: chars, shorts, unsigned shorts are automatically promoted to int.
Rule 2: When an operator works with two values of different data types, the lower data type is promoted to a higher data type.

- ✓ **Type Casting**
  Type casting means to convert a higher data type to a lower data type. For the purpose of type casting we require to use the type case operator, which lets you manually promote or demote a value. It is a unary operator which appears as the data type name followed by the operand inside a set of parentheses. E.g.

```
val = (int)number;
```

Operator's Precedence & Associativity Table

Operator	Description	Associativity
( ) [ ] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.,* **z++** evaluates to **z**) and **z** only incremented after all else is done. See **postinc.c**for another example.

Conditional Program Execution :

Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in C programs using a decision control instruction.

- C language has decision making capabilities and supports controlling of statements.
- C supports following control or decision making statements:

1. " if " Statements.
2. " switch " Statements.

1. If statement :

- The keyword **if** tells the compiler that what follows is a decision control instruction.
- The condition following the keyword **if** is always enclosed within a pair of parentheses.
- If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed;
- **if** statement is a powerful decision making statement.
- **if** statement is used to control the flow of execution of statements.

Different forms of If statement

1.1 Simple If statement.

1.2 If…..else statement.

1.3 Nested if…..else statement.

1.4 Else if ladder or else if …else .

1.1 Simple if statement

- If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed;
- The general form of a simple if statement is :

Syntax =>

```
if (Condition)
        {
                True statement 1;
                True statement 2;

                ...
        }
```

Ex : WAP in C to Say Hello if no. is positive.

```
int main( )
{
        int x ;
        printf("Enter Value For X : ");
        scanf("%d", &x);
        if(x>=0)
        {
                printf("Hello User!!!");
        }
        return 0;
}
```

1.2 If….Else statement

- IF the if expression evaluates to true, the block following the if statement or statements are executed.
- The else statement is optional. It is used only if a statement or sequences of statements are to be executed in case, the if expression evaluates to false.
- The group of statements after the **if** upto and not including the **else** is called an 'if block'. Similarly, the statements after the **else** form the 'else block'.
- Notice that the **else** is written exactly below the **if**.

Syntax =>

```
if(Condition)
{
  True statement…1
  True statement…2
  …
}
else
{
  False statement 1
  False statement 1
  …
}
```

Ex : WAP in C to check no. is even or odd.

```
int main( ) {
        int x ;
        printf("Enter Value For X : ");
        scanf("%d", &x);
        if(x%2 == 0)
        {
                printf("X is EVEN number.");
        }
        else
        {
                printf("X is ODD number.");
        }
        return 0;
}
```

1.3 Else if ladder

- The if – else – if statement is also known as the if-else-if ladder or the if-else-if staircase.
- The conditions are evaluated from the top downwards.

Syntax:-

```
if(condition)
{
  True statement…1
  True statement…2
  …
}
else if(condition)
{
  True statement…1
  True statement…2
  …
}
else
{
  False statement 1
  False statement 1
  …
}
```

Ex : WAP in C to find max among three no.

```
int main( ) {
        int x = 20, y = 27, z = 10 ;
        if( x>y  &&  x>z )
        {
                printf("X is GREATER");
        }
        else if( y>z )
        {
                printf("Y is GREATER");
        }
        else
        {
                printf("Z is GREATER");
        }
        return 0;
}
```

1.4 Nested if…else statement:

- It is perfectly all right if we write an entire **if-else** construct within either the body of the **if** statement or the body of an **else** statement. This is called 'nesting 'of **if**s.
- The general form of nested if…else statement is:

```
Syntax:
if (condition 1)
 {
        if (condition 2)
        {
                statement 1;
        }
        else
        {
                statement 2;
        }
 }
else
 {
        statement 3;
 }
```

Ex : WAP in C to find max among three no.

```
int main( ) {
        int  x = 20, y = 27, z = 10 ;
        if( x>y )
        {
                if( x>z )
                  printf("X is GREATER");
                else
                  printf("Z is GREATER");
        }
        else
        {
                if( y>z )
                  printf("Y is GREATER");
                else
                  printf("Z is GREATER");
        }
        return 0;
}
```

Example : In a company an employee is paid as under: If his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary. If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

```
void main( )
{
        float bs, gs, da, hra ;
        printf ( "Enter basic salary : " ) ;
        scanf ( "%f", &bs ) ;
        if ( bs < 1500 )
         {
                hra = bs * 10 / 100 ;
                da = bs * 90 / 100 ;
        }
        else
        {
                hra = 500 ;
                da = bs * 98 / 100 ;
        }
         gs = bs + hra + da ;
         printf ( "Gross salary = Rs. %f ", gs ) ;
}
```

2. switch statement :

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch-case-default**, since these three keywords go together to make up the control statement. They most often appear as follows:

```
switch ( integer expression )
{
        case constant 1 :
                do this ;
        case constant 2 :
                do this ;
        case constant 3 :
                do this ;
        default :
                do this ;
}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others.

- ✓ **Execution Of switch :**
  First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case**

- ✓ **default with switch :**
  If no match is found with any of the **case** statements, only the statements following the **default** are executed.

Consider the following program:

```
int  main( )
{
     int i = 2 ;
     switch ( i )
     {
          case 1 :
               printf ( "I am in case 1 \n" ) ;
          case 2 :
               printf ( "I am in case 2 \n" ) ;

          case 3 :
               printf ( "I am in case 3 \n" ) ;

          default :
               printf ( "I am in default \n" ) ;
     }
     return 0;
}
```

The output of this program would be:
I am in case 2
I am in case 3
I am in default

The output is definitely not what we expected! We didn't expect the second and third line in the above output. The program prints case 2 and 3 and the default case.

- ✓ **break with switch**

  Well, yes. We said the **switch** executes the case where a match is found and all the subsequent **cases** and the **default** as well.

  If you want that only case 2 should get executed, it is upto you to get out of the **switch** then and there by using a **break** statement. The following example shows how this is done. Note that there is no need for a **break** statement after the **default**, since the control comes out of the **switch** anyway.

Consider the following program:

```
int main( )
{
     int i = 2 ;
     switch ( i )
     {
          case 1 :
               printf ( "I am in case 1 \n" ) ;
               break;
          case 2 :
               printf ( "I am in case 2 \n" ) ;
               break;
          case 3 :
               printf ( "I am in case 3 \n" ) ;
               break;
          default :
               printf ( "I am in default \n" ) ;
     }
     return 0;
}
```

The output of this program would be:
I am in case 2

Program Loops and Iteration :

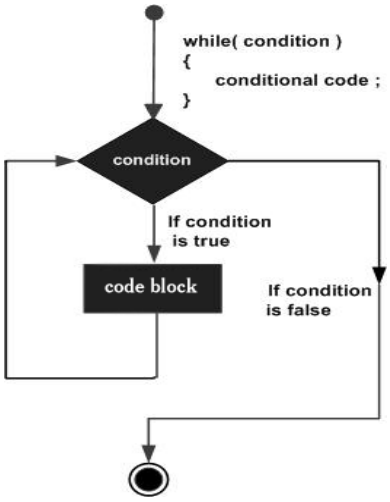
The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction.

There are three types of loop

1. while
2. for
3. do…while

1. while Loop :

The ***while*** statement creates a loop that repeats until the test ***expression*** becomes false, or zero. The ***while*** statement is an entry-condition loop; the decision to go through one or more loop is made before the loop is traversed. Thus, it is possible that the loop is never traversed. The ***statement*** part of the form can be a simple statement or a compound statement.



Syntax :

```
while(condition)
{
    Statement 1;
    ……
}
```

Example :	Output :

#include<stdio.h>
void main ()
{
  int a = 10;
 	
  while(a < 20)
  {
  printf("value of a: %d\n", a);
  a++;
  }
 	
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 15
	value of a: 16
	value of a: 17
	value of a: 18
	value of a: 19

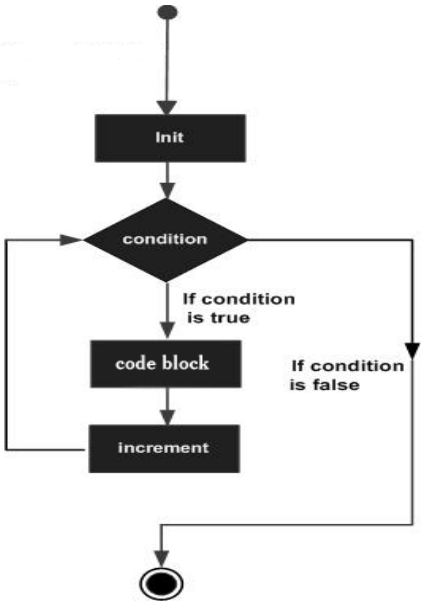
2. for Loop :

- ✓ A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
- ✓ Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

```
for( init, condition, increment)
{
        Statement 1;
        ……
}
```

Here is the flow of control in a for loop:



1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Example :	Output :

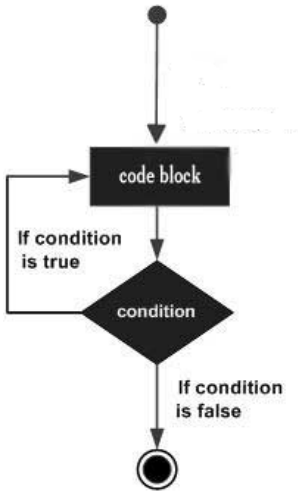
#include <stdio.h>
void main ()
{
  int a;
  for(a = 10; a < 20; a = a + 1)
  {
  printf("value of a: %d\n", a);
  }
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 15
	value of a: 16
	value of a: 17
	value of a: 18
	value of a: 19

3. do…while Loop :

- ✓ Like a while statement, except that it tests the condition at the end of the loop body.
- ✓ Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.
- ✓ A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

```
do
{
        Statement1;
        ……
}while( condition);
```



Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Example :	Output :

#include <stdio.h>
void main ()
{
int a = 10;
 	
do
{
printf("value of a: %d\n", a);
a = a + 1;
} while(a < 20);
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 15
	value of a: 16
	value of a: 17
	value of a: 18
	value of a: 19

❖ **Difference Between while and do-while loop :**

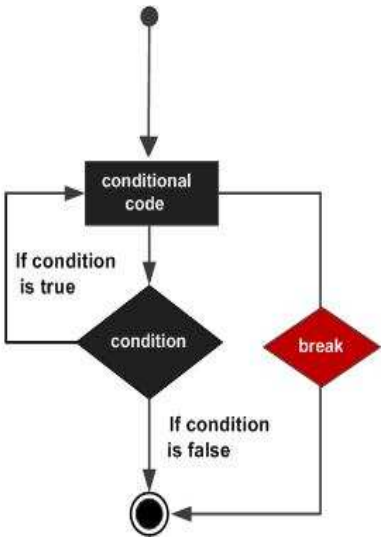
Difference between while and do-while loop	
while	**do…while**
1. while loop is entry control loop	1. do while is exit control loop
2. In while loop the condition is tested first, if the condition is true then statements are executed.	2. In do while the statements are executed first and then the condition are tested to execute again.
3. while loop do not run in case the condition given is false.	3. A do while loop runs at least once.
4. In a while loop the condition is first tested and if it returns true then it goes in the loop.	4. A do while is used for a block of code that must be executed at least once.

5. Syntax:
while(condition)
{
statement(s);
}
	5. Syntax:
	do
	{
	satatement(s);
	}while(condition);

❖ **break statement :**

The **break** statement in C programming language has following two usage:

- ✓ When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- ✓ It can be used to terminate a case in the **switch** statement.



If you are using nested loops (i.e. one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Example :	Output :

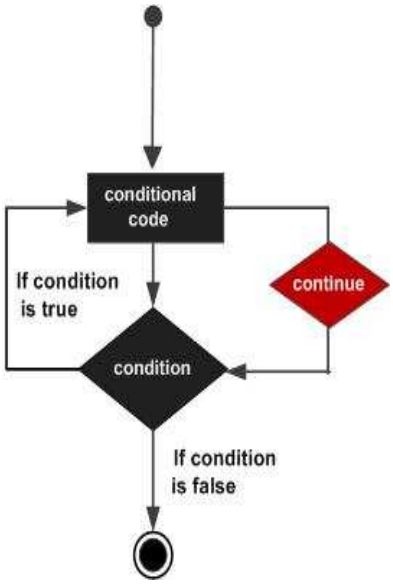
#include <stdio.h>
int main () {
int a = 10;
 	
while(a < 20)
{
printf("value of a: %d\n", a);
a++;
if(a > 15)
{
break;
}
}
return 0;
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 15

❖ **continue statement :**

- ✓ The **continue** statement in C programming language works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.
- ✓ For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For he **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests

Example :	Output :

#include <stdio.h>
int main () {
int a = 10;
do
{
if(a == 15)
{
a = a + 1;
continue;
}
printf("value of a: %d\n", a);
a++;
}while(a < 20);
return 0;
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 16
	value of a: 17
	value of a: 18
	value of a: 19



❖ **goto statement :**

- ✓ A **goto** statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.
- ✓ **NOTE:** Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

Example :	Output :

#include <stdio.h>
int main () {
int a = 10;
display:
printf("value of a: %d\n", a);
a++;
if(a <= 15)
{
goto display;
}
return 0;
}
	value of a: 10
	value of a: 11
	value of a: 12
	value of a: 13
	value of a: 14
	value of a: 15

C FUNCTION

Modular Programming :

Modular programming is a software design technique that break downs a program into individual components(modules) that can programmed and tested independently. It is a requirement for effective development and maintenance of large programs and projects.

Advantages of modular programming:

i)Manageable : Reduce problem to smaller, simpler, humanly comprehensible problems.

ii)Divisible : Modules can be assigned to different teams/programmers. It enables parallel work thereby reducing program development time. Also it facilitates programming, debugging, testing and maintenance.

iii)Re-usable : Modules can be re used within a program and across programs.

In C language, modules are implemented as functions.

Functions

- ✓ A function is a group of statements that together perform a task. Every C program has at least one function which is **main()**, and all the most trivial programs can define additional functions.

- ✓ You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

- ✓ Types Of Functions :
  1. **Library Function:** The C standard library provides numerous built-in functions that your program can call. For example, function **printf()** to display the formatted output on screen.

  2. **User Define Function:** If a set of statements are needed to execute more than one times then user can define their own function and call when needed. For example, function **add(int a , int b)** to add two integers a and b.

❖ Function Prototype :

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax:

Return-type function-name (parameter list);

Example :

int add(int a, int b);

Parameter names are not important in function declaration(prototype) only their type is required, so following is also valid declaration

int add(int, int);

❖ Function Definition :

The general form of a function definition in C programming language is as follows:

```
Return-type   function-name(parameter-list)
{
        Body of function;
}
```

A function definition in C programming language consists of a *function header* and a *function body*. Here are all the parts of a function:

- ✓ **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

- ✓ **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- ✓ **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- ✓ **Function Body:** The function body contains a collection of statements that define what the function does.

Example :

```
int add(int a , int b)
{
        int sum;
        sum = a + b;
        return sum;
}
```

❖ Function Calling :

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value.

Example : `int result = add(20, 30);`

EXAMPLE :

```
#include <stdio.h>

// function prototype
int getCube(int);

int main ( )
{
   int a,cube;

   printf("Enter Value For a : ");
   scanf("%d", &a);

// function calling
   cube = getCube(a);

   printf("Cube of a is %d.", cube);
   return 0;
}

//function definition
int getCube(int x)
{
     int c;
     c = x*x*x;
     return c;
}
```

OUTPUT :

Enter Value For a : 3
Cube of a is 27.

Scope Rules :

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called **local** variables,
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which is called **formal** parameters.

❖ Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main( ) function.

```
#include <stdio.h>

int main ()
{
  /* local variable declaration */
  int a, b;
  int c;

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

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

  return 0;
}
```

❖ Global Variables

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

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
/* global variable declaration */
int g;

int main ()
{
  /* local variable declaration */
  int a, b;

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

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

  return 0;
}
```

❖ **Formal Parameters**

A function parameters, formal parameters, are treated as local variables with-in that function and they will take preference over the global variables. Following is an example:

```
/* global variable declaration */
int a = 20;

int main ()
{
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

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

  return 0;
}

/* function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by address	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max( ) function used the same method.

Note : Call By Reference will discuss with pointer chapter

❖ Call By Value :

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses *call by value* method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

Example : Consider the function **swap( )** definition as follows.

```
#include <stdio.h>

 /* function declaration */
void swap(int , int );

int  main ( )
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}

/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
}
```

OUTPUT :

Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200

❖ **Recursion :**

- ✓ Recursion is the process by which a function calls itself from itself. In other words, the function call is embedded in the function definition.
- ✓ The function body has a terminating condition which if satisfied terminates further call to itself and the remaining statements if any in the previous call are executed.
- ✓ Thus recursion follows a bottom-up approach, where the last function call is executed first, followed by its immediate previous call and so on.
- ✓ The advantage of recursion is that it reduces program code, thus making it simpler.

Example :

```
/*FACTORIAL OF n BY RECURSION USING USER DEFINE FUNCTION*/

#include<stdio.h>

int fact(int);
int  main ( )
{
      int num,factorial;
      int fact(int n);
      printf("Enter the value :");
      scanf("%d",&num);
      factorial=fact(num);
      printf("The factorial of %d is : %d",num, factorial);
      return 0;
}
int fact(int n)
{
      if(n==1)
      return 1;
      else
      return(n*fact(n-1));
}
```

OUTPUT :

Enter the value : 4
The factorial of 4 is : 24

Storage Classes :

To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class'.

There are basically two kinds of locations in a computer where such a value may be kept— Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:

- Where the variable would be stored.
- What will be the initial value of the variable, if initial value is not specifically assigned.(i.e. the default initial value).
- What is the visibility of the variable; i.e. in which functions the value of the variable would be available.
- What is the scope or life of the variable; i.e. how long would the variable exist.

There are four storage classes in C:

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

(a)Automatic storage class : (this is default class for local variables)

Keyword : auto
Default value : garbage
Storage : memory
Scope of variable : Local to block
Life of variable : till control remains in the block in which it is declared.

Ex: auto int x;

Ex:

```
#include<stdio.h>          //header file
int main()                 //main function
{
  auto int x=10;
  printf("%d",x);
  return 0;
}
```

Output :20 ,10

(b)Register storage class :

Keyword : register
Default value : garbage
Storage : registers of CPU
Scope of variable : Local to block
Life of variable : till control remains in the block in which it is declared.

Ex: register int x;

```
int  main()
{
     register int r;
     for(r=1;r<=10;r++)
     Printf("%d",r);
     return 0;
}
```

Output:1,2,3,4,5,6,7,8,9,10

Note : If any CPU register is not available to store data then 'c' compiler automatically convert the storage class from register to automatic, because number of CPU registers are limited..

(c)Static storage class

Keyword : static
Default value : zero
Storage : memory
Scope of variable : local to block
Life of variable : till the program is running, and it persist in various function call

Ex: static int x;

```
int main()
{
    static int x;
    printf("%d",x);
    x++;
    if(x<=10)
    main();
    return 0;
}
```

Output:0,1,2,3,4,5,6,7,8,9,10

(4)Extern storage class :

Keyword : extern
Default value : zero
Storage : memory
Scope of variable : global from point of declaration onwards
Life of variable : till program execution does not come to an end
Ex: extern int x;

```
int x=10;
int main()
{
   extern int x;
   printf("%d",x);
   return 0;
}
```

Output: 10

ARRAYS AND STRINGS

Arrays in C :

C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of **an array as a collection of variables of the same type.**

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of **contiguous** memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Key Points : Array

- ✓ An array is a collection of similar data items and these data types may be all ints, floats or all chars.
- ✓ Each member of an array is identified by unique index assigned to it.
- ✓ An index is a positive integer enclosed in [ ] placed immediately after the array name.
- ✓ An index holds integer value starting with zero. Means the first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- ✓ An array is defined in the same way as a variable defined except that array name is followed by one or more expressions, enclosed within square brackets[ ], specifying the array dimension.
- ✓ The general syntax of an array is:

  Data type array_name[size];

  Example: int marks[30];
- ✓ The data type specifies the type of elements that will be contained in the array.
- ✓ The array name indicates the location of the first member of an array.
- ✓ The array size indicates the maximum number of elements stored in the array.

- **Declaring Arrays**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

```
double balance[10];
```

Now *balance* is avariable array which is sufficient to hold upto 10 double numbers.

- ***Initializing Arrays***

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

- ***Accessing Array Elements***

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
   int n[ 10 ]; /* n is an array of 5 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 5; i++ )
   {
     n[ i ] = i + 100; /* set value at loc. i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 5; j++ )
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }
  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
```

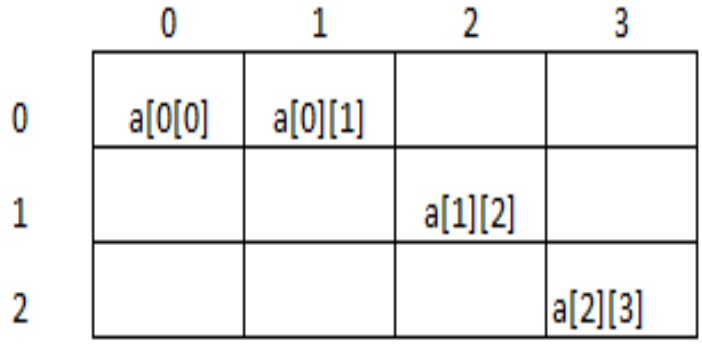
In C there are mainly three types of array :-

Single dimensional array.

```
Ex: int x[3]={10,20,30};
```

Double dimensional array.

```
Ex:- int a[3][4]={ {1,2,3,4},{1,2,3,4},{4,3,2,1} };
```



Multi dimensional array.

```
Ex:- int x[2][3][2]={
{{10,20},{30,40},{50,60}},{{11,22},{33,44},{55,66}} };
```

Example of two dimensional array.

```
int a[3][4];
```

➢ Accessing Two -Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts ie. row index and column index of the array.
For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

void main ()
{
   /* an array with 4 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 4; i++ )
   {
      for ( j = 0; j < 2; j++ )
      {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }
}
```

When the above code is compiled and executed, it produces following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
```

Strings Manipulation:

A string is an array of characters which is terminated by a null character ('\0').

- ✓ Each character in an array occupies one byte of memory and the last character is always "\0" (Backslash Zero) .
  For example:- **char name[]={'H','E','W','L','E','T','T','\0'};**
- ✓ A string variable is any valid C variable name and is always declared as an array.
- ✓ The general syntax of string variable is:
- ✓ When the compiler assigns a character string to character array, it automatically supplies a null character('\0') at the end of the string.
- ✓ Therefore the size of string must be equal to the maximum number of characters plus one.
- ✓ Following are the examples for initialization of string.
  - o **char name[12]="HPES India";**
  - o **char name[5]={'H','P','E','S','\0'};**
- ✓ C also permits us to initialize a character array without specifying the number of elements, for example:-
  char str[]={'H','E','L','L','O','\0'};

Ex:-/*To print the characters using string */

```
#include<stdio.h>
int main()
{
     int i;
     char name[10];
     printf( "Enter your name");
     scanf( "%s", &name);
     printf("\n Hello %s",name);
     return 0;
}
```

Output:

```
Enter your name Santosh
Hello Santosh
```

- C library supports a wide range of string handling functions, which are found in standard header file **<string.h>**
- Some of the string handling functions are:-

❖ strcat( )

- The strcat function joins two string together. It copies string 2 to string 1

  Syntax: strcat(string1,string2);

Ex:
```
char a[10]=”Hai”;
char b[10]={‘B’,’y’,’e’,’\0’};
strcat(a,b);
printf(“%s”,a);
```
Output:->HaiBye

- C also permits nesting of strcat functions, which concatenates all the string in to one string such as.

  Syntax: strcat((string1,string2),string 3);

❖ strcmp( ) :

- The strcmp function compares two string to find out whether they are same or different.
- Where string are compared character by character until there is a mismatch or end of one of the strings is reached.
- This function returns 0 if both strings are same, positive (greater than zero) values if string1 is bigger, negative (less than zero) value if string 1 is small then string 2.
- Syntax: strcmp((string1,string2)
- Ex:1:-> strcmp(“RCPL”,”RCPL”) **output-> 0**
  strcmp(“RAM”,”ROM”) **output-> -14**
  strcmp(“ROM”,”RAM”) **output-> 14**

❖ strcpy( )

- The strcpy function copies the contents of one string into another.
- The base address of the source and target string should be supplied to this function.

Ex:
```
char a[10]=”India”;
char b[10]=”Bharat”;
strcpy(a,b);
prinf(“%s”,a);
```
Output:->Bharat

❖ strlen( )

- The strlen function counts and returns the number of characters in a string(not count null char '\0').

```
int x=strlen("India");
printf("%d",x);
```

output:-> 5

❖ strrev( )

- reverse the given string

Ex:->
```
char c[10]="United";
strrev(c);
printf("%s",c);
```

output:-> detinU

POINTERS

Pointers in C :

A pointer is a variable whose value is the address of another variable ie. direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

❖ How to use Pointers?

There are few important operations which we will do with the help of pointers very frequently.
(a) we define a pointer variables
(b) assign the address of a variable to a pointer and
(c) finally access the value at the address available in the pointer variable.
This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>
int main () {
   int  var = 20;    /* actual variable declaration */
   int  *ip;         /* pointer variable declaration */

   ip = &var;        /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

  /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

➢ NULL Pointers in C

It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
The value of ptr is 0
```

➢ Pointer Arithmetic:

C pointer is an address which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –

✓ Incrementing a Pointer

Incrementing a pointer, which increases its value by the number of bytes of its data type as shown below:

```
int x = 30;          //Suppose Address of X is 200
int *p  =  &x;       //Suppose Address of P is 100
p++;                 //Increment the address of P by 2 i.e. 202 because size of int is 2 bytes
```

✓ Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
int x = 30;          //Suppose Address of X is 200
int *p  =  &x;       //Suppose Address of P is 100
p--;                 //Decrement the address of P by 2 i.e. 198 because size of int is 2 bytes
```

✓ Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

- ✓ Following arithmetic operations are valid with pointers:-

```
p1= p1+ 4;
p1= p1-2;
p3= p1-p2;
```

- ✓ Following other operations are valid with pointers:-

```
p1>p2
p1==p2
p1 != p2
p1++;
--p2;
```

- ✓ Following operations are not valid with pointers:-

```
p3=p1+p2;
p3=p1/p2;
p3=p1*p2;
p1=p1/3;
```

➢ **Using Pointers as function argument :** Call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp = *x;     /* save the value at address x */
   *x = *y;           /* put y into x */
   *y = temp;         /* put temp into y */
 }
```

```
#include <stdio.h>
void swap(int *x, int *y); // function prototype
int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(&a, &b);  //calling swap by passing addresses of a and b
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap,  value of a :200
After swap,  value of b :100
```

- **Difference between call by value & call by reference.**

Cal by Value	Call by Reference
This is the usual method to call a function in which only the value of the variable is passed as an argument	In this method, the address of the variable is passed as an argument
Any alternation in the value of the argument passed is local to the function and is not accepted in the calling program	Any alternation in the value of the argument passed is accepted in the calling program(since alternation is made indirectly in the memory location using the pointer)
Memory location occupied by formal and actual arguments is different	Memory location occupied by formal and actual arguments is same and there is a saving of memory location
Since a new location is created, this method is slow	Since the existing memory location is used through its address, this method is fast

POINTER to POINTERS :

- ✓ In C programming pointers are variables that stores address of another memory space and they have their own address.

  Eg: int x = 20; //suppose the address of **x** is 65524
  int *p = &x; //**p**'s value will be 65524 and address is 20024

- ✓ If we create a pointer that stores an address of another pointer variable that type of pointer is called **pointer to pointer** (double pointer).

  Eg: int x = 20;
  int *p1 = &x;
  int **p2 = &p1;

  ACCESS CODES:

  p2 : access the value of p2 that is address of p1 (20024)
  * p2 : access the value of p1 that is address of x (65524)
  ** p2 : access the value of x (20)

POINTER to FUNCTOIN :

- ✓ The C programming function also loads into the memory space to store the definition of function for execution.
- ✓ As we know **Pointers** are variable that can store memory address, so C programming allows to store the function address in pointers, such pointers are called **function pointers.**
- ✓ Function pointers can be used to do almost anything you can do with a function name.

EXAMPLE : 1

```
int main(){
  void (*f) = disp;
  f();
  return 0;
}
void disp(){
 printf("DISPLAY CALLED!!");
}
```

OUTPUT : DISPLAY CALLED!!

EXAMPLE : 2

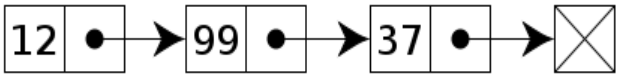
```
int main(){
  void (*f)(int,int) = add;
  f(20,30);
  return 0;
}
void add(int a, int b){
  int sum = a+b;
  printf("ADDITION RESULT IS : %d",sum);
}
```

OUTPUT:

ADDTION RESULT IS : 50

Linked List :

- ✓ A **linked list** is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence.
- ✓ Linked lists are among the most common data structures, and are used to implement many important data structures, such as stacks, queues.
- ✓ Linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.
- ✓ Linked lists contain nodes which have a data field as well as a *next* field, which points to the next node in the linked list.



❖ Linked List Operations :

We can perform various operations on such a list. The most common operations are:

- *checking* whether the list is empty;
- ***inserting*** or ***removing*** a specific element (e.g., the first one, the last one, or one with a certain value);
- ***accessing*** a node to modify it or to obtain the information in it;
- ***traversing*** the list to access all elements (e.g., to print them, or to find some specific element);
- determining the ***size*** (i.e., the number of elements) of the list;

Dynamic Memory Allocation :

Sometime applications(software) requires allocation of memory at runtime as per the requirement or need, so C programming have a provision for dynamic memory allocation on heap memory area using some library function.

The C programming language provides several functions for memory allocation and management. These functions can be found in the **<stdlib.h>** or **<alloc.h>** header file.

S.N.	Function and Description
1.	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size**.
2.	**void free(void *address);** This function release a block of memory block specified by address.
3.	**void *malloc(int num);** This function allocates an array of **num** bytes and leave them initialized.
4.	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize**.

The functions malloc and calloc obtain blocks of memory dynamically.

- ✓ **`void *malloc(size_t n)`**
  returns a pointer to n bytes of uninitialized storage, or NULL if the request cannot be satisfied.
- ✓ **`void *calloc(size_t n, size_t size)`**
  returns a pointer to enough free space for an array of n objects of the specified size, or NULL if the request cannot be satisfied. The storage is initialized to zero.

The pointer returned by malloc or calloc has the proper alignment for the object in question, but it must be cast into the appropriate type, as in

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

STRUCTURE & UNION

Structures in C :

- ✓ Structure is collections of items of different data types.
- ✓ Similar to array, for structure also contiguous memory space for all elements.
- ✓ Structure's elements are called **member** of structure.

Structures are used to represent a record, Suppose you want to keep track of your students in a college You might want to track the following attributes about each student:

- Roll Number
- Name
- Branch

➢ Defining a Structure

To define a structure, you must use the **struct** keyword. The format of the struct statement is given below:

```
struct structure_name
{
   member detail;
   ...
   member detail;
} [one or more structure variables];
```

In structure each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Student structure:

```
struct Student
{
   int rollno;
   char  name[30];
   char  branch[20];
}s1, s2;
```

- **Accessing Structure Members : Normal variable**

  To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure:

```
#include <stdio.h>
struct Student {
   int rollno;
   char  name[30];
   char  branch[20];
};

int main( ) {
  struct Student s1;
  printf("Enter Student Details : ");
  printf("ROLL NO : ");
  scanf("%d",s1.rollno);
  printf("NAME    : ");
  gets(&s1.name);
  printf("BRANCH  : ");
  gets(&s1.branch);

  printf("-----------------------\n");
  printf("ROLL NO  : %d\n",s1.rollno);
  printf("NAME     : %s\n",s1.name);
  printf("BRANCH   : %s\n",s1.branch);
  printf("-----------------------\n");

  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
-----------------------
ROLL NO  : 100651
NAME     : Ranjan Verma
BRANCH   : MECHENCAL
-----------------------
```

- **Accessing Structure Members : Pointer to structure**

  To access any member of a structure with pointre, we use the **member access operator (->)**. The member access operator is coded as a period between the structure's pointer variable name and the structure member that we wish to access. You would use **struct** keyword to define pointer variables of structure type. Following is the example to explain usage of structure:

```
#include <stdio.h>
struct Student {
   int rollno;
   char  name[30];
   char  branch[20];
};

int main( ) {
  struct Student *s1;
  printf("Enter Student Details : ");
  printf("ROLL NO : ");
  scanf("%d",s1->rollno);
  printf("NAME    : ");
  gets(&s1->name);
  printf("BRANCH  : ");
  gets(&s1->branch);

  printf("-----------------------\n");
  printf("ROLL NO  : %d\n",s1->rollno);
  printf("NAME     : %s\n",s1->name);
  printf("BRANCH   : %s\n",s1->branch);
  printf("-----------------------\n");

  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
-----------------------
ROLL NO  : 100663
NAME     : Mohd Ayaan
BRANCH   : ELECTRICAL
-----------------------
```

Union in C :

- ✓ A **union** is a special user defined data type available in C that allows to store different data variables in the same memory location.
- ✓ A ***union*** is a memory location that is shared by two or more variable generally of different types at different time.
- ✓ Thus a ***union*** contains many members of different types, but it can handle only one member at a time.
- ✓ A ***union*** is defined in the same way as structure is defined but in place of **struct** keyword here we use **union** keyword, the general format is:

```
union union_name {
  Member1 details;
  Member2 details;
  :::::::::
};
```

In union each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. We can use only one member at a time because union allocates common memory space for all members.

```
union demo {
    int i;
   float j;
}ob;
```

In the above code , C will allocate sizeof(float) bytes memory for **ob** that will be common for **i and j** , so we can use only last updated value i.e. once at a time.

- **Accessing Union Members : Normal variable**

  To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the **union** variable name and the **union** member that we wish to access. You would use **union** keyword to define variables of **union** type. Following is the example to explain usage of **union**:

```
#include <stdio.h>
union demo {
   int i;
   int j;
};

int main( ) {
  union demo ob;
  ob.i = 40;
  ob.j = 50;
  printf("------------------------\n");
  printf("VALUE OF I  : %d\n",ob.i);
  printf("VALUE OF J  : %d\n",ob.j);
  printf("------------------------\n");

  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
------------------------
VALUE OF I  :  50
VALUE OF J  :  50
------------------------
```

- **Accessing Union Members : Pointer to Union**

  To access any member of a union, we use the **member access operator (->)**. The member access operator is coded as a period between the **union** variable name and the **union** member that we wish to access. You would use **union** keyword to define variables of **union** type. Following is the example to explain usage of **union**:

```
#include <stdio.h>
union demo {
   int i;
   int j;
};

int main( ) {
  union demo *ob;
  ob->i = 40;
  ob->j = 50;
  printf("------------------------\n");
  printf("VALUE OF I  : %d\n",ob->i);
  printf("VALUE OF J  : %d\n",ob->j);
  printf("------------------------\n");

  return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
------------------------
VALUE OF I  :  50
VALUE OF J  :  50
------------------------
```

Nested Structure:

- ✓ **Nested structure** in C is nothing but structure within structure. One structure can be declared inside other structure as we declare structure members inside a structure.

```
struct Student {
  int rollno;
  char name[30];
  char branch[20];

  struct date {
    int d;                          //to store date
    int m;                          //to store month
    int y;                          //to store year
  };

  struct date dob;
};
```

Self Referential Structure:

- ✓ **Self referential structure** in C is a structure that have a pointer variable that can point another structure of same type.
- ✓ A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.

```
struct Student {
  int rollno;
  char name[30];
  char branch[20];

  struct Student *bestfriend;
};
```

Difference Between Structure and Union:

Difference between Structure and Union	
Structure	**Union**
1.The keyword **struct** is used to define a	1. The keyword **union** is used to define a
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared to all members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset	4. The address is same for all the members of a union. This indicates that every member begins at the same
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.

FILE HANDLING

File Handling

- A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind.
- A file can be a text file or a binary file depending upon its contents.
- Through file handling, one can perform operations like create, modify, delete etc on system files.
- File I/O can be performed on a character by character basis, a line by line basis, a record by record basis or a chunk by chunk basis.
- Special functions have been designed for handling file operations. The header file **stdio.h** is required for using these functions.

File Handling function's

- In C language, we use a structure pointer of file type to declare a file.

  FILE *fp;

- C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	description
fopen()	create a new file or open an existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads an integer from a file
putw()	writes an integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

C File Operation: -

There are five major operations that can be performed on a file:

- Creation of a new file.
- Opening an existing file.
- Reading data from a file.
- Writing data in a file.
- Closing a file.

Opening a File or Creating a File

The **fopen ()** function is used to create a new file or to open an existing file.

fopen ()returns the address of the file, which we have collected in the file pointer called **fp**. We have declared **fp** as

FILE *fp;

General Syntax:

fP = fopen(" file name " , "Opening Mode");

OR

***fp** = FILE ***fopen** (const char ***filename**, const char ***mode**);

Here **filename** is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

***fp** is the FILE pointer **(FILE *fp)**, which will hold the reference to the opened (or created) file.

The mode can be:

mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode

wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The **fclose()** function is used to close an already opened file.

General Syntax :

int **fclose** (FILE***fp**);

Here **fclose ()** function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h.**

fputc( ) :-> Writes a single character to a file.

```
Syntax:-fputc( char , FILE *);
Ex:-> char ch='a';
fputc(ch,fp);
```

fgetc( ) :- > Reads a single character from a file.

```
Syntax:-char fgetc(FILE *);
Ex:- > char ch=fgetc(fp);
```

fputs( ) :-> Writes a string in to file.

```
Syntax:- fputs(const char *,FILE *)
Ex:- fputs("C was developed in 1972",fp);
```

fgets ( ) :→ Reads a string from a file.

```
Syntax:- char * fgets(char *, int, FILE *)
Ex:-> char ch[50];
ch=fgets(fp);
```

fread( ) :-> Read data from a file

fwrite( ):-> Writes data to a file

ftell( ) :

Function *ftell()* returns the current position of the file pointer in a stream. The return value is 0 or a positive integer indicating the byte offset from the beginning of an open file. A return value of -1 indicates an error.

```
long int ftell(FILE *fp);
```

fseek( )

This function positions the next I/O operation on an open stream to a new position relative to the current position.

```
int fseek(FILE *fp, long int offset, int origin);
```

Here *fp* is the file pointer of the stream on which I/O operations are carried on, *offset* is the number of bytes to skip over. The *offset* can be either positive or negative, denting forward or backward movement in the file. *origin* is the position in the stream to which the offset is applied, this can be one of the following constants:

SEEK_SET: offset is relative to beginning of the file
SEEK_CUR: offset is relative to the current position in the file
SEEK_END: offset is relative to end of the file

EOF: It is a constant indicating that **End of File** has been reached in a file.

Ex :-char by char write into file

```
#include<stdio.h>
int main()
{
      FILE *p;
      char ch;
      p=fopen("d:\\FH\\abc.txt","w");
      do
       {
      printf("Enter a char '@' to exit = ");
      scanf("%c",&ch);
      fputc(ch,p);
       }while(ch!='@');
      fclose(p);
}
```

Ex: - Write and read character one by one into a file

```
#include<stdio.h>
int main()
{
   char ch;
   FILE *fp;
   fp=fopen(“test.txt”,”w”);
     do
         {
          Printf(“enter charater to write into file or ‘@’ to exit..”)
            ch=getchar();
            fputc(ch,fp);
        }while(ch!=’@’);
    fclose(fp);
         fp=fopen(“test.txt”,”r”);
     do
        {
           ch=fgetc(fp);
           printf(“%c”,ch);
       }while(ch!=EOF);
        fclose(fp);
     }
     }
```

Ex:- To Copy content of one file ”test.txt” to another file “aa.txt”

```
     #include<stdio.h>
     int main()
     {
        char ch;
        FILE *fp, *fq;
        fp=fopen(“test.txt”,”r”);
     fq=fopen(“aa.txt”,”w”);

     do
         {
            ch=fgetc(fp);
            fputc(ch,fq);
     }while(ch!=EOF);
     fclose(fp);
       fclose(fq);
     }
```

ADDITIONAL FEATURES IN C

Command Line Argument :

- Sometimes it is useful to pass information into a program when you run it. Generally, you pass information into the **main( )** function via command line arguments.
- A *command line argument* is the information that follows the program's name on the command line of the operating system.

```
int main(int argc, char *argv[])
```

- Two special built-in arguments, **argc** and **argv**, are used to receive command line arguments. The **argc** parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The **argv** parameter is a pointer to an array of character pointer(i.e. String).

Example:

```
int main(int argc, char *argv[])
{
  if(argc!=2)
  {
     printf(''You forgot to type your name.\n");
  }
  printf("Hello %s", argv[1]);
  return 0;
}
```

NOTE : Source file name is test.c

OUTPUT :

```
Compile : gcc test.c
Run     : ./a.out
     You forgot to type your name.
Run     : ./a.out India
     Hello India
```

Enumerated Data Types:

- An *enumeration* is a set of named integer constants. Enumerations are common in everyday life. For example, an enumeration of the day is Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.
- Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type. The general form for enumerations is

```
enum  tag{enumeration list};
```

Example: WAP to check given number is prime or not.

enum boolean {false, true};
*/*here false and true are actually two constant integer values that is false=0 and true=1 by default */*

```
int main( ){
      int num, i;
      enum boolean flag = false;
      printf("Enter any value : ");
      scanf("%d",&num);
      for(i=2; i<=num/2; i++) {
          if(num%i == 0)  {
               flag =  true;
          }
     }
      if(flag == false)
          printf("%d is PRIME NUMBER",num);
      else
          printf("%d is NOT PRIME NUMBER",num);
}
```

```
OUTPUT:1
Enter any value : 6
6 is NOT PRIME NUMBER

OUTPUT:2
Enter any value : 11
11 is NOT PRIME NUMBER
```

C Preprocessors:

✓ The C preprocessor is exactly what its name implies. It is a program that processes our source program before it is passed to the compiler.

✓ The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:

(a) Macro expansion
(b) File inclusion → #include<stdio.h> etc..
(c) Conditional Compilation .

Macro

❖ The **#define** directive defines an identifier and a character sequence (a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*

Have a look at the following program.

```
#define UPPER 25
int main( )
{
      int i ;
      for ( i = 1 ; i <= UPPER ; i++ )
      printf ( "\n%d", i ) ;
      return 0;

}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement, **#define UPPER 25**

➢ Macros with Arguments

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
        float r1 = 6.25, r2 = 2.5, a ;
        a = AREA ( r1 ) ;
        printf ( "\nArea of circle = %f", a ) ;
        a = AREA ( r2 ) ;
        printf ( "\nArea of circle = %f", a )
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **(3.14 * x * x)**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **(3.14 * x * x)**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**.

Thus the statement **AREA(r1)** is equivalent to: (3.14 * r1 * r1)

C Preprocessors : contd..

➢ Conditional Compilation

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef  macroname
        statement 1 ;
        statement 2 ;
#endif
```

If **macroname** has been **#define**d, the block of code will be processed as usual; otherwise not.

```
int main( )  {
      #ifdef INTEL
            code suitable for a Intel PC
      #else
            code suitable for a Motorola PC
      #endif
            code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C.

❖ *#if , #else and #endif* Directives

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped.

A simple example of **#if** directive is shown below:

```
main( )  {
      #if  TEST <= 5
            statement 1 ;
            statement 2 ;
            statement 3 ;
      #else
            statement 4 ;
            statement 5 ;
            statement 6 ;
      #endif
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled.

Standard C Library Functions :

C provide a set of built in functions in their library for different purpose, those function are declared in header files and their definition are in C library that link with our code at the time of creation of executable code by linker.
The standard library provides a wide variety of functions.

❖ Mathematical Functions

There are more than twenty mathematical functions declared in <math.h>; here are some of the more frequently used. Each takes one or two double arguments and returns a double.

sin(x)	sine of x, x in radians
cos(x)	cosine of x in radians
exp(x)	exponential function e^x
pow(x,y)	x^y
sqrt(x)	square root of x (x>0)
abs(x)	absolute value of x

❖ Character Class Testing and Conversion

Several functions from <ctype.h> perform character tests and conversions. In the following, c is an int that can be represented as an unsigned char or EOF. The function returns int.

isalpha(c)	non-zero if c is alphabetic, 0 if not
isupper(c)	non-zero if c is upper case, 0 if not
islower(c)	non-zero if c is lower case, 0 if not
isdigit(c)	non-zero if c is digit, 0 if not
isalnum(c)	non-zero if isalpha(c) or isdigit(c), 0 if not
isspace(c)	non-zero if c is blank, tab, newline, return, formfeed, vertical tab
toupper(c)	return c converted to upper case
tolower(c)	return c converted to lower case

PRACTICE SET

PRACTICE ALGORITHMS

1. ADDITION OF GIVEN TWO NUMBER.

STEP	1	:	START	
STEP	2	:	A=10, B=20	
STEP	3	:	C=A+B	
STEP	4	:	PRINT “SUM” C	
STEP	5	:	STOP	

2. ADDITION OF GIVEN TWO NUMBER.

STEP	1	:	START	
STEP	2	:	INPUT A	
STEP	3	:	INPUT B	
STEP	4	:	C=A+B	
STEP	5	:	PRINT “SUM” C	
STEP	6	:	STOP	

3. SIMPLE INTEREST OF GIVEN PRINCIPAL, RATE AND TIME.

STEP	1	:	START	
STEP	2	:	INPUT P, R, T	
STEP	3	:	SI=P*R*R/100	
STEP	4	:	PRINT “SIMPLE INTERST” SI	
STEP	5.	:	STOP	

4. FIND OUT AREA AND PERIMETER OF GIVEN RADIUS OF CIRCLE

STEP	1	:	START	
STEP	2	:	INPUT R	
STEP	3	:	PERI=2*3.14*R	
STEP	4	:	AREA=3.14*R*R	
STEP	5	:	PRINT “PERIMETER” PERI	
STEP	6	:	PRINT “AREA”, AREA	
STEP	7	:	STOP	

5. WRITE A ALGORITM TO SWAPING OF GIVEN TWO NUMBER

STEP	1	:	START	
STEP	2	:	INPUT A, B	A=10 B=20
STEP	3	:	C=A	C=A=10
STEP	4	:	A=B	A=B=20
STEP	5	:	B=C	B=C=10
STEP	6	:	PRINT A, B	A=20 B=10
STEP	7	:	STOP	

6. WRITE A ALGORITM TO SWAPING OF GIVEN TWO NUMBER WITHOUT THIRD VIRIABLE

STEP	1	:	START	
STEP	2	:	INPUT A, B	A=10 B=20
STEP	3	:	A=A+B	A=10+20=30
STEP	4	:	B=A-B	B=30-20=10
STEP	5	:	A=A-B	A=30-10=20
STEP	6	:	PRINT A, B	A=20 B=10
STEP	7	:	STOP	

General Expression	C Expression
(A+B)(C+D)	(A+B)*(C+D)
(A+B)/(C+D)	(A+B)/(C+D)
AX^2 +BX+X=0	A*X*X+B*X+C=0
$X=2^3$	X=POW(2,3)
Area= $\sqrt{s(s-a)(s-b)(s-c)}$	Area=SQRT(S*(S-A)*(S-B)*(S-C))
S=A+B+C/3	S=(A+B+C)/3

7. FIND OUT THE AREA OF TRIANGLE GIVEN THREE SIDES

STEP	1	:	START
STEP	2	:	INPUT A, B, C
STEP	3	:	S= (A+B+C)/2
STEP	4	:	AREA=SQRT(S*(S-A)*(S-B)*(S-C))
STEP	5	:	PRINT “AREA” AREA
STEP	6	:	STOP

NOTE: C/5= (F-32)/9
F=32 FIND THE VALUE OF C THE EXPRESSION MUST BE C=5*(F-32)/9
C=05 FIND THE VELUE OF F THE EXPRESSION MUST BE F=9*C/5+32

8. FIND THE POWER GIVEN TWO NUMBER 2^3=8

STEP	1	:	START
STEP	2	:	INPUT A, B
STEP	3	:	C=POW (A, B)
STEP	4	:	PRINT C
STEP	5	:	STOP

9. IMPLEMENT THE FORMULA $A=P(1+R/100)^N$

STEP 1 : START
STEP 2 : INPUT P, R, N
STEP 3 : S= 1+R/100
STEP 4 : A=P*POW(S, N)
STEP 5 : PRINT "AMOUNT" A
STEP 6 : STOP

10. DISPLAY ALL NATURAL NUMBER 1 TO 10 1 2 3 4 5 6 7 8 9 10

STEP 1 : START
STEP 2 : COUNT = 0
STEP 3 : COUNT = COUNT +1
STEP 4 : PRINT COUNT
STEP 5 : IF COUNT < = 10 THEN GOTO STEP 3 ELSE STEP 6
STEP 6 : STOP

11. DISPLAY ALL EVEN NUMBER 1 TO 10

STEP 1 : START
STEP 2 : COUNT = 0
STEP 3 : COUNT = COUNT +2
STEP 4 : PRINT COUNT
STEP 5 : IF COUNT < = 10 THEN GOTO STEP 3 ELSE STEP 6
STEP 6 : STOP

12. WRITE AN ALGORITHM DISPLAY 1 TO N NUMBER

STEP 1 : START
STEP 2 : COUNT = 0
STEP 3 : INPUT "HOW MANY STEPS" N
STEP 4 : COUNT =COUNT+1
STEP 5 : PRINT COUNT
STEP 6 : IF COUNT < =N THEN GOTO STEP 4 ELSE STEP 7
STEP 7 : STOP

13. DISPLAY THE SUM OF ALL NATURAL NUMBER 1 TO 10

STEP 1 : START
STEP 2 : COUNT = 0, SUM = 0
STEP 3 : COUNT = COUNT +1
STEP 4 : SUM=SUM+COUNT
STEP 5 : IF COUNT < 10 THEN GOTO STEP 3 ELSE STEP 6
STEP 6 : PRINT "SUM", SUM
STEP 7 : STOP

14. DISPLAY THE SUM OF GIVEN NUMBER 1 TO 10

STEP	1	:	START		
STEP	2	:	COUNT = 1, SUM = 0		
STEP	3	:	SUM=SUM+N		
STEP	4	:	COUNT = COUNT + 1		
STEP	5	:	IF COUNT <= 10 THEN GOTO STEP 3 ELSE STEP 6		
STEP	6	:	PRINT “SUM”, SUM		
STEP	7	:	STOP		

15. FIND OUT GIVEN NUMBER IS ODD OR EVEN

STEP	1	:	START		
STEP	2	:	INPUT “ANY NUMBER” N		
STEP	3	:	M=N%2		
STEP	4	:	IF M==0 THEN GOTO STEP 5 ELSE GOTO STEP 7		
STEP	5	:	PRINT “EVEN”		
STEP	6	:	GOTO STEP 8		
STEP	7	:	PRINT “ODD”		
STEP	8	:	STOP		

16. FIND OUT REVERS NUMBER GIVEN 3 DIGIT NUMBER

STEP	1	:	START		
STEP	2	:	INPUT “ANY 3 DIGIT NUMBER” N	123	
STEP	3	:	REV=0		0
STEP	4	:	M=N%10		3
STEP	5	:	N=N/10		12
STEP	6	:	REV=REV*10+M	3	
STEP	5	:	M=N%10		2
STEP	6	:	N=N/10		1
STEP	7	:	REV=REV*10+M	32	
STEP	8	:	M=N%10		1
STEP	9	:	N=N/10		0
STEP	10	:	REV=REV*10+M	321	
STEP	11	:	PRINT “REVERSE NUMBER” REV		
STEP	12	:	STOP		

17. FIND OUT SUM OF DIGIT GIVEN 3 DIGIT NUMBER

STEP	1	:	START		
STEP	2	:	INPUT “ANY 3 DIGIT NUMBER” N	123	
STEP	3	:	SUM=0		0
STEP	4	:	M=N%10		3
STEP	5	:	N=N/10		12
STEP	6	:	SUM=SUM+M		3
STEP	5	:	M=N%10		2
STEP	6	:	N=N/10		1
STEP	7	:	SUM=SUM+M		5
STEP	8	:	M=N%10		0
STEP	9	:	N=N/10		1
STEP	10	:	SUM=SUM+M		6
STEP	11	:	PRINT “REVERSE NUMBER” SUM		
STEP	12	:	STOP		

18. FIND OUT EQUAL NUMBER OR NOT EQUAL NUMBER GIVEN TWO NUMBER

STEP 1 : START
STEP 2 : INPUT “ENTER TWO NUMBER” A, B
STEP 3 : IF A==B THEN GOTO STEP 4 ELSE GOTO STEP 6
STEP 4 : PRINT “EQUAL NUMBER”
STEP 5 : GOTO STEP 7
STEP 6 : PRINT “NOT EQUAL NUMBER” B
STEP 7 : STOP

19. FIND OUT LARGEST NUMBER GIVEN TWO NUMBER

STEP 1 : START
STEP 2 : INPUT “ENTER TWO NUMBER” A, B
STEP 3 : IF A>B THEN GOTO STEP 4 ELSE GOTO STEP 6
STEP 4 : PRINT “MAXIMUM NUMBER” A
STEP 5 : GOTO STEP 7
STEP 6 : PRINT “MAXIMUM NUMBER” B
STEP 7 : STOP

20. FIND OUT GIVEN NUMBER IS “POSSITIVE” , “NEGATIVE” OR “EQUAL TO ZERO”

STEP 1 : START
STEP 2 : INPUT “ENTER ANY NUMBER” A
STEP 3 : IF A>0 THEN GOTO STEP 4 ELSE GOTO STEP 6
STEP 4 : PRINT “POSITIVE NUMBER”
STEP 5 : GOTO STEP 7
STEP 6 : IF A<0 THEN GOTO STEP 7 ELSE GOTO STEP 9
STEP 7 : PRINT “NEGATIVE NUMBER”
STEP 8 : GOTO STEP 10
STEP 9 : PRINT “EQUAL TO ZERO”
STEP 10 : STOP

21. FIND OUT THE FACTORIAL GIVEN ANY NUMBER

STEP 1 : START
STEP 2 : INPUT “ENTER ANY NUMBER” NUM
STEP 3 : FACT=1
STEP 4 : COUNT=0
STEP 5 : COUNT=COUNT+1
STEP 6 : FACT=FACT*COUNT
STEP 7 : IF COUNT<=NUM THEN GOTO STEP 5 ELSE GOTO STEP 8
STEP 8 : PRINT “FACTORAL” FACT
STEP 9 : STOP

22. FIND OUT THE AVERAGE OF GIVEN NUMBER OF GIVEN N NUMBER

```
STEP 1 : START
STEP 2 : INPUT "ENTER HOW MANY NUMBER" N
STEP 3 : COUNT = 0
STEP 4 : SUM = 0
STEP 5 : REAPEATE WHILE COUNT <  N
         a) : INPUT "ENTER ANY NUMBER" NUM
         b) : SUM = SUM + NUM
         c) : COUNT = COUNT + 1
         WHILE END
STEP 6 : AVG = SUM / N
STEP 7 : PRINT "AVERAGE" AVG
STEP 8 : STOP
```

FIBONACII NUMBER

0 1 1 2 3 5 8 13 21

23. WRITE AN ALGORITHM TO PRINT THE FIRST 100 FIBONACII NUMBER

```
STEP 1  : START
STEP 2  : INPUT "HOW MANY STEPS" N
STEP 3  : A=0
STEP 4  : B=1
STEP 5  : PRINT A, B
STEP 6  : COUNT =2
STEP 7  : COUNT = COUNT +1
STEP 8  : C=A+B
STEP 9  : PRINT C
STEP 10 : A=B
STEP 11 : B=C
STEP 12 : IF COUNT < N THEN GOTO STEP 7 ELSE GOTO STEP 13
STEP 13 : STOP
```

24. W RITE AN ALGORITHM TO SUM THE FIRST 100 FIBONACII NUMBER AND

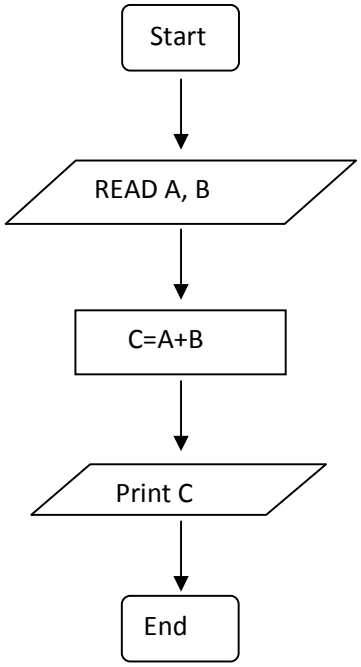
```
STEP 1  : START
STEP 2  : INPUT "HOW MANY STEPS" N
STEP 3  : A=0
STEP 4  : B=1
STEP 5  : PRINT A, B
STEP 6  : SUM = A+B
STEP 7  : COUNT =2
STEP 8  : COUNT = COUNT +1
STEP 9  : C=A+B
STEP 10 : PRINT C
STEP 11 : SUM=SUM+C
STEP 12 : A=B
STEP 13 : B=C
STEP 14 : IF COUNT < N THEN GOTO STEP 7 ELSE GOTO STEP 13
STEP 15 : PRINT SUM
STEP 16 : STOP
```

25. FIND OUT REVERS NUMBER GIVEN ANY NUMBER

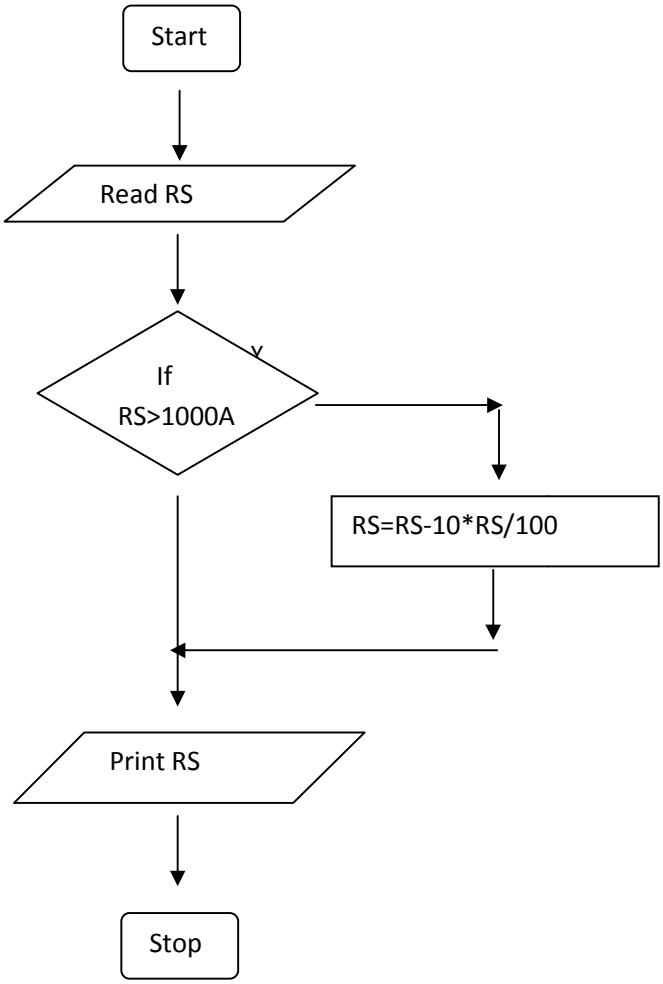
STEP	1	:	START				
STEP	2	:	INPUT “ENTER ANY NUMBER” N	123			
STEP	3	:	REV=0		0		
STEP	4	:	M=N%10		3	2	1
STEP	5	:	N=N/10		12	1	0
STEP	6	:	REV=REV*10+M	3	32	321	
STEP	5	:	IF N>0 THEN GOTO STEP 4 ELSE GOTO STEP 6				
STEP	6	:	PRINT “REVERSE NUMBER” REV				
STEP	7	:	STOP				

PRACTICE FLOW CHART

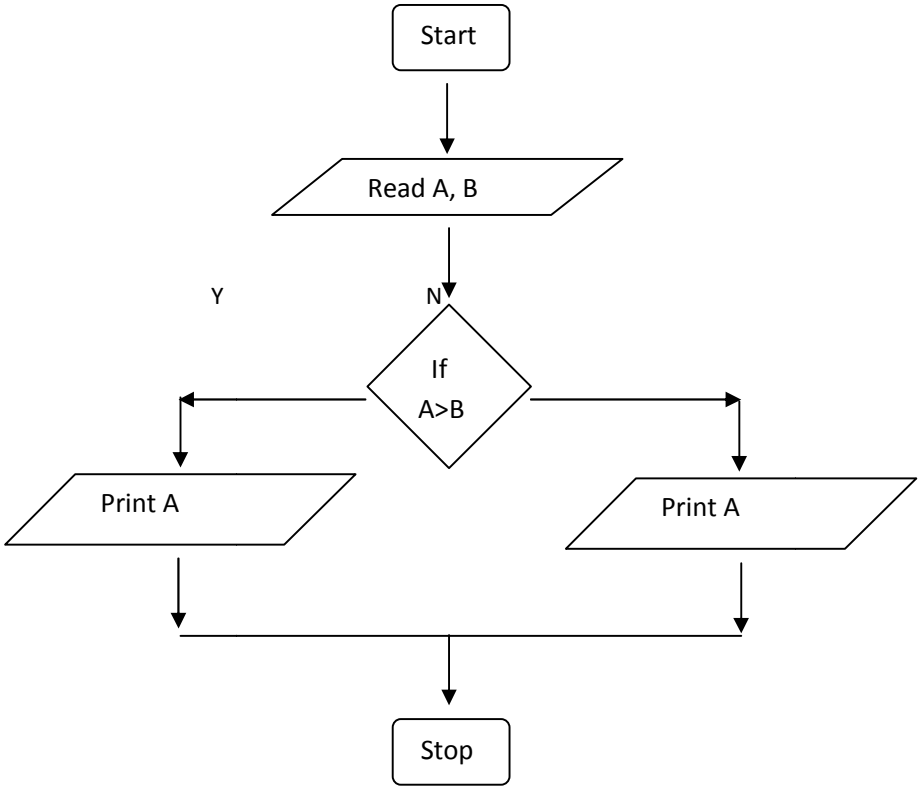
1 DRAW A FLOW CHART ACCEPT TWO NUMBER FIND OUT THE SUM



2 DRAW A FLOW CHART GIVEN A DISCOUNT 10% IF PURCHASING MORE THE RS. 1000. OTHER WISE NO ANY DISCOUNT.



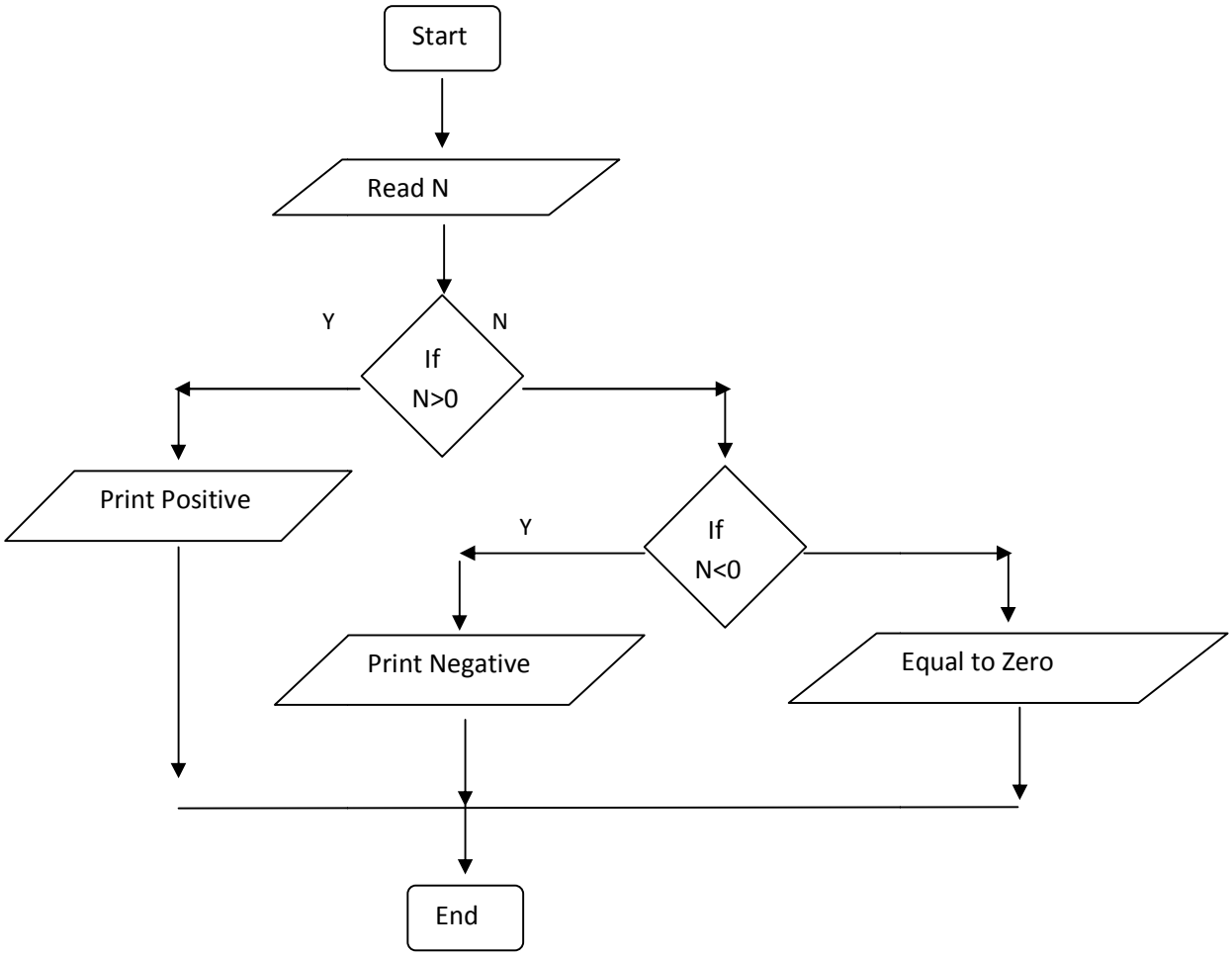
3 DRAW A FLOW CHART ACCEPT TWO NUMBER FIND OUT THE MAXUMUM NUMBER



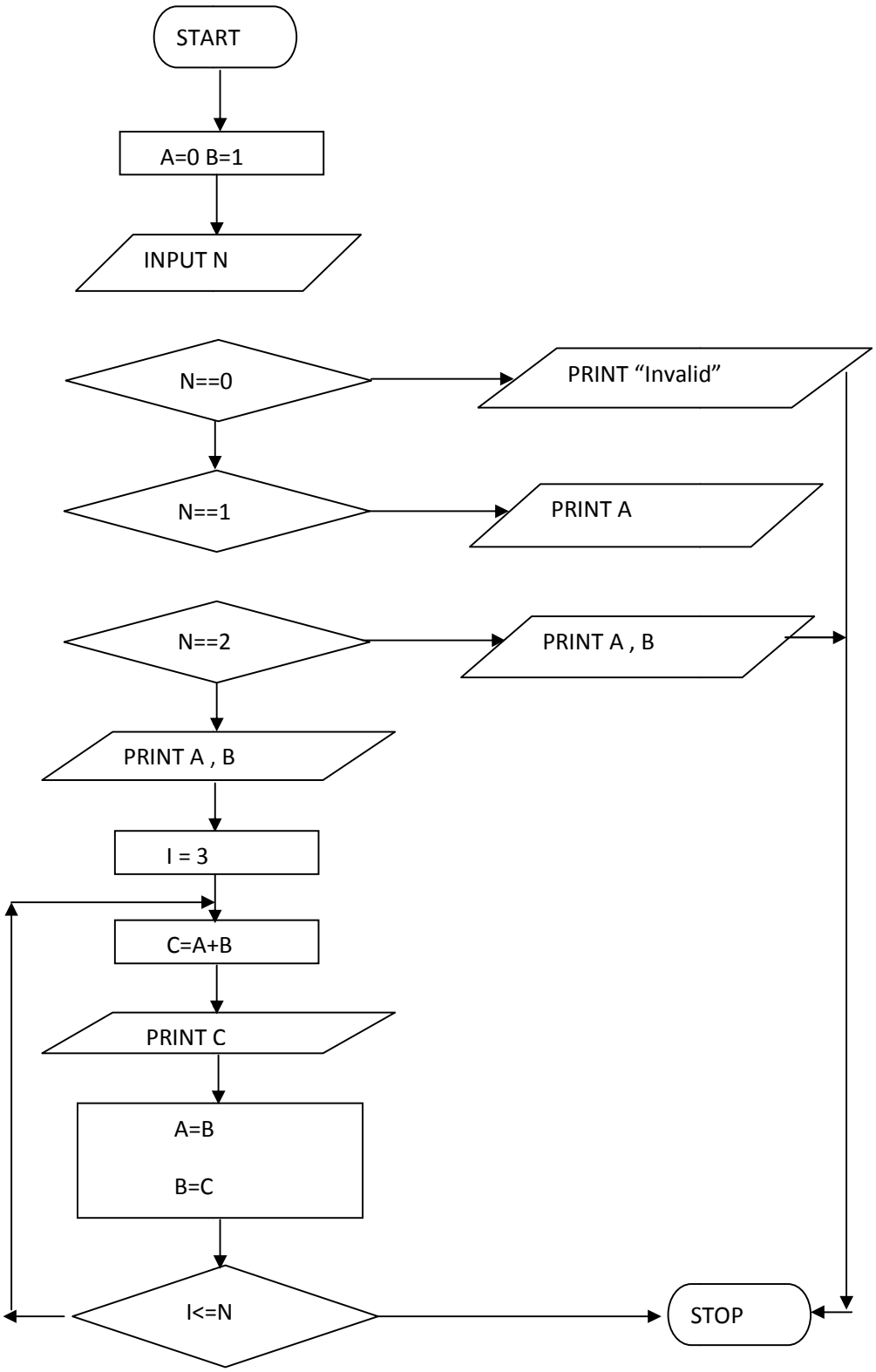
4 MAKE A FLOW CHART FOR FINDING MAXIMUM NUMBER OUT OF ANY GIVEN THREE NUMBER. **(for self practice)**

5 DRAW A FLOW CHART FOR FINDING OUT LARGEST OF GIVEN THREE POSITIVE NUMBER **(for self practice)**

6 **DRAW A FLOW CHART ACCPET ANY NUMBER FIND OUT THE GIVEN NUMBER IS "POSSITIVE NUMBER" , "NEGATIVE NUMBER" OR "EQUAL TO ZERO"**



7. DRAW A FLOW CHART TO PRINT THE FIBONACII SERIES GIVEN 'N' NUMBER



PRACTICE PROGRAMS

1. Write a program(WAP) to find the sum of the digits of a number.
2. WAP to check the given number is **palindrome** or not.
3. WAP to check the given number is **prime** or not.
4. WAP to find the factorial of a number using function.
5. WAP to check weather the given number is **perfect number** or not.
6. WAP to print **Fibonacci** series upto **n**th item.
7. WAP to find and display the **area** of a circle of given radius.
8. WAP to print all **even** numbers from 1 to 100.
9. WAP to print the **sum** of numbers from 1 to 15
10. WAP to find the **sum** of given array elements.
11. WAP to find the **maximum** and **minimum** element of given array.
12. WAP to print the array elements in **reverse** order.
13. WAP to display the **transpose** of 3X3 matrix using **2D int array**.
14. WAP to find **sum** of two 3X3 matrix using **2D int array**.
15. WAP to find **factorial** of a given number using **recursion** function.
16. WAP to print **Fibonacci** series using **recursion** function.
17. WAP to find the given number is **Armstrong** or not.
18. WAP to find the sum of the following series :

    1 + 3 + 5 + 7 + 9 + ……. n

19. Write a C program to print the following pattern :

    1
    2 6
    3 9 27
    4 12 36 108

20. WAP to count all positive and negative numbers in given array.
21. WAP to take text as in input from keyboard and print its **reverse**.
22. Write a C program to print the following pattern :

    1
    2 2
    3 3 3
    4 4 4 4

23. WAP to take a number from keyboard and print its binary conversion.
24. WAP to count the number of zeros (0) in 3X3 integer array.
25. WAP to input details of 10 account and print only names of those account holder whose balance amout is greater than 50000 using structure.