MODULE-1

CHAPTER 1: INTRODUCTION TO COMPUTER HARDWARE AND SOFTWARE

1. INTRODUCTION

Computer is an electronic device which accepts input, processes data, stores information and produces output.

Data: Raw facts/ figures

Information: Processed data

2. COMPUTER GENERATIONS

The computer of each generation is smaller, faster and more powerful than preceding generation. There are five computer generations.

- **First Generation:** The vacuum tubes were used for computation. Magnetic drums were used for memory requirements. It consumed lot of space, power and generated lot of power. ENIAC (Electronic Numerical Integrator and Computer) used 18000 vacuum tubes, 1800 acquired sq. ft. room space and consumed 180KW of power. The machine level language (0s and 1s) was used. Punched cards were used for input and Paper for output. They were used for scientific work.

- **Second Generation:** The transistors was the most important component which replaced vacuum tubes. Magnetic cores were used for memory. It were more reliable than first generation computer. The assembly or symbolic language was used. The input and output mechanism remained same. The stored program concept was introduced which stores both data and program.

- **Third Generation:** The Integrated circuits(IC) was the most important component. The transistors, diodes, resistors, capacitors were integrated on a single chip. The high-level language was used like BASIC, C, C++ and JAVA. Memory capacity increased and magnetic hard disk was used for secondary generation. The third generation computers also had OS and computer could rum programs invoked by multi users.

- **Fourth Generation:** The Microprocessor was the most important component. With the help of LSI (Large Scale Integration) and VLSI (Very Large Scale Integration) the entire CPU is on a single chip. OS have moved from MSDOS to GUI (Graphical User Interface) like windows. The networking technology has also been improved. The size was reduced and the speed was increased.

- **Fifth Generation:** Artificial Intelligence and use of natural languages are the main features of this generations. These systems are expected to interact with users in natural language. Speech recognition and speech output should also be possible. Computers must be able to perform parallel processing. The quad-core and octa-core was also introduced. Neural networks and expert systems have been developed.

3. COMPUTER TYPES

Apart from being classified by generations, computers can also be categorized by their size. The size of a computer is often an indirect indication of its capabilities.

➢ **Supercomputers:** These are huge machines having most powerful and fastest processors. It uses multiple CPUs for parallel data processing. Speeds are measured in flops (floating point operations per second). The fastest operates at 34 petaflops. They are used for weather forecasting, analysis of geological data. They have enormous storage, uses more power and generate lot of heat. They are used by government agencies.

➢ **Mainframes:** These are multi-user machines that support many users using the feature of time sharing. It can run multiple programs even with a single CPU. The processor speed is measured in MIPS (Million instructions per second). It is used to handle data, applications related to organization and online transactions in banks, financial institutions and large corporations.

➢ **Minicomputers/Midrange computers:** It was introduced by DEC (Digital Equipment Corporation). They can serve hundreds of users and are small enough to partially occupy a room. They are used in smaller organizations or a department of a large one. They are not affordable to be used in home.

➢ **Microcomputers:** The microcomputer or PC is introduced by Apple and endorsed by IBM. This is a single-user machine powered by a single-chip microprocessor. They are very powerful machines having gigabytes of memory. They are both used in standalone mode and in a network. A microcomputer takes the form of desktop, notebook (laptop) or a netbook (smaller laptop). PCs today are powered by 3 types of OS – windows (7, 8 or 10), Mac OS X (Apple) and Linux. They are used for engineering and scientific applications and for software development.

➢ **Smartphones and Embedded Computers:** The smartphone is a general purpose computer i.e., capable of making phone calls. It has a powerful processor, with multiple cores, supports GBs of memory, and runs developed OS (Android or iOS). It can be operated with keyboard, touch or stylus.

Embedded Computers or micro-controllers are very small circuits containing a CPU, non-volatile memory, input and output handling facilities. They are embedded into many machines that we use – cars, washing machines, cameras etc. The processor here runs a single unmodifiable program stored in memory.

4. BITS, BYTES AND WORDS

- Computer can understand only two states: 0 and 1.
- A digit can have only two states or values known as a binary digit, abbreviated as bit (b).
- The name nibble was coined to represent four bits.
- The name byte (B) was coined to represent eight bits.
- The byte is the standard unit of measurement of computer memory, data storage and transmission speed.
- The CPU handles memory data in larger units, called words and it is usually even multiple of bytes (two bytes, four bytes etc.).
- When referred to a computer it has 32-bit (4 bytes) machine i.e., size of word is 32 bits.

Unit	Equivalent to	Remarks
1 kilobyte (KB)	1024 bytes	Space used by 10 lines of text
1 megabyte (MB)	1024 kilobytes	Memory of the earliest PCs
1 gigabyte (GB)	1024 megabytes	Storage capacity of a CD-ROM
1 terabyte (TB)	1024 gigabytes	Capacity of today's hard disks
1 petabyte (PB)	1024 terabytes	Space used for rendering of film Avatar

5. CPU (CENTRAL PROCESSING UNIT)

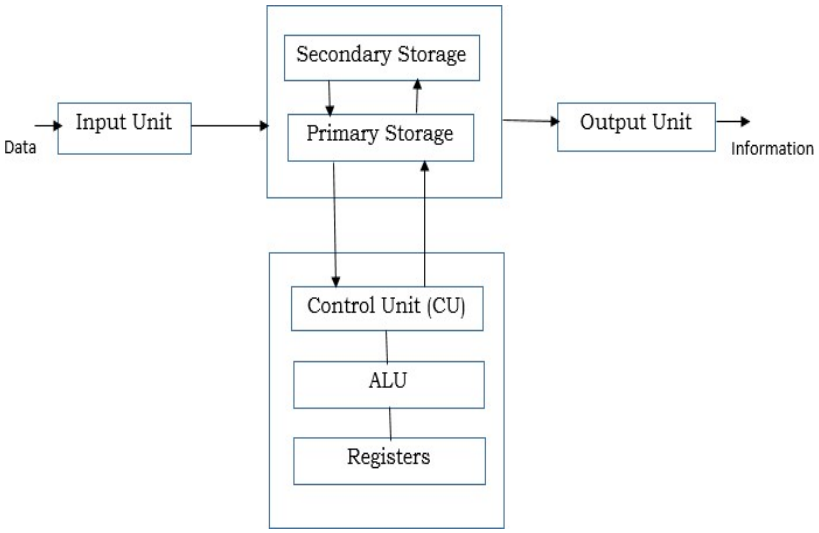


Figure 1: Functional Components of Computer

The CPU comprises of following components:

i. ALU (Arithmetic and Logic Unit)
ii. CU (Control Unit)
iii. Special-purpose registers
iv. A clock

- The *ALU* is a "super calculator" carrying out all arithmetic tasks and Boolean operations.
- The *CU* controls the way data is moved between the various components of computer.
- Both ALU and CU use the service of *clock* for synchronizing their Operations.
- The CPU uses a few high-speed *registers* to store the current instruction and its data. One of the registers, program counter, stores the address of next instruction to be executed.
- All program instructions are executed using the *fetch-decode-execute mechanism.*
- The CPUs are rated in GHz (gigahertz).
- *Input Unit:* It is used to give the input that is data to the computer. It is done with the help of input devices. Ex: Mouse, Keyboard
- *Output Unit:* It is used to receive the output (information) from the computer. It is done with the help of output devices. Ex: Printer, Speaker.
- *Memory:* Collection of chips on motherboard, where all computer processing and program instructions are stored. There are two types: Primary Memory and Secondary Memory.

6. PRIMARY MEMORY

Primary memory is the main memory, which stores data and programs, which are currently needed by CPU. The size is less than the secondary memory and it is costly

1. **RAM (Random Access Memory):** It is the main memory, where the user can write information into RAM and read information from it. It is accessible to users. The RAM is randomly accessible by user. It is a volatile memory, which means the data, or information is retained as long as power supply is ON. There are two types of RAM: SRAM and DRAM.

SRAM (Static)	DRAM (Dynamic)
Stores information as long as power supply is on, reloads every 2ms.	Loses data in a very short time.
Multiple transistors are used to store 1 bit.	One transistor is used to store 1 bit.
It is expensive, faster, and bigger.	It is inexpensive, slower, and smaller.
It requires more power.	It requires less power.
It is not required to be refreshed.	It requires refresh.

2. **ROM (Read Only Memory):** It is a permanent memory that can be read but not written. It is a nonvolatile memory, which means the data or information is retained even power supply is not there. It contains a startup program BIOS (Basic Input Output System) which transfers control to OS.

   - **PROM (Programmable Read Only Memory):** It is programmed as per requirement of customer's choice. The programmer burns the data into PROM. The data once written cannot be changed.

   - **EPROM (Erasable Programmable Read Only Memory):** It can be rewritten (but only once) even though it has been previously burned. It is erased by exposing it to UV-rays.

   - **EEPROM (Electrically Erasable Programmable Read Only Memory):** It can be erased and rewritten multiple times. The electric voltage is used to erase the data. The pen drive we use today is of EEPROM.

3. **Cache Memory:** It holds those portions of program that are frequently used by CPU. It acts as a buffer between CPU and RAM. The CPU first looks for the instructions in cache. It executes faster than RAM, expensive and limited in size. It has multiple levels:
   - **L1 (Level 1)** – smallest and fastest – 32 KB
   - **L2 (Level 2) –** present closer to CPU – 256 KB
   - **L3 (Level 3) –** shared by cores – 8 MB

4. **Registers:** The small number of ultra-fast registers integrated into the CPU represent the fastest memory of the computer. The CPU does all its work here. Each register has the length of the computer. The data is loaded into register before processing. Registers are numbered and a program instruction specifies these numbers. Ex: OR1R2R3 which means multiplication operation is performed on R1 and R2 is performed and stored in R3.

7. SECONDARY MEMORY

Secondary memory is not directly connected to CPU. It exists inside the machine and also externally. It is a non-volatile, offline and long-term storage memory. It is slower, cheaper than primary memory but the capacity is higher.

1. **Hard disk/ Hard drive/ Fixed disk:** It is the oldest secondary storage device. It has more capacity but, the cost is less comparatively. It is commonly present in laptop with 500GB and desktop with 1TB. It contains a spindle, which holds one or more platters made up of non-magnetic material. It has two surfaces which is coated with magnetic material. Each surface has serially numbered tracks and further broken into sectors or blocks. The disk runs with the speed of 5400 and 7200 rpm.

2. **Magnetic tape:** The magnetic tape is made up of plastic film with one side coated with magnetic material. It supports 1 TB or more, but 200 TB are also expected. The device is not fully portable though because, a separate tape drive is required. The data is accessed sequentially. This makes it unsuitable for backup.

3. **Optical Disks (CD-ROM, DVD ROM, Blu-ray Disk):** It is non-volatile read-only memory. The CD-ROM and DVD-ROM, can hold large volumes of data (700MB to 8.5 GB). The Blu-ray disk has the capacity upto 50 GB. A laser beam in their drives controls the read and write operations. A laser beam is used to construct pits and lands by burning selected areas along its tracks.
   CD-R, DVD-R: Data can be recorded only once.
   CD-RW, DVD-RW: Data can be recorded multiple times.

4. **Flash Memory:** It doesn't have any moving parts, is based on the EEPROM. It is available in various forms- pen drive, magnetic card (SD Card), solid state disk (SSD). They are portable, need little power and quite reliable.
   The **memory stick or pen drive** is the most common type of flash memory used on the computer. It is a small, removable piece of circuit and it connects to the USB port of computer.
   The **solid state disk** is a bigger device meant to replace the traditional magnetic hard disk.
   The **magnetic card** is used mainly in cameras and the most popular is the micro-SD card.

5. **Floppy Diskette:** This is represented by a rectangular plastic case containing a thin magnetic disk. It was available in two sizes (5.25" and 3.5"), offering capacities of 1.2 MB and 1. MB. It is unsuitable for backup purpose.

✓ **Difference between Primary memory and Secondary memory**

Primary Memory	Secondary Memory
The size of the primary memory is small.	The size of the secondary memory is larger.
Stores the programs and data currently needed by the CPU.	It is used to store large amount of data and programs
Expensive.	Inexpensive.
Volatile in nature.	Non-volatile in nature.
Ex: RAM, ROM	Ex: Hard disk, Optical disk

8. PORTS AND CONNECTORS

Devices like scanners, printers are connected to a computer through docking points called ports. It is impossible to use wrong connector for a port. At present systems offer only fewer types compared to old ones.

➢ **USB (Universal Serial Bus):** This replaced serial and parallel ports in motherboard. Most computers offer four USB ports to support scanners, printers and mice. It has four lines, two each for data and power.

The current USB 3.0 can transfer 1 GB file in 20 seconds. The smaller variant, the micro-USB port is used in smartphones.

- **SERIAL PORT:** This port transfers one bit at a time serially. They are offered in 9-pin and 25-pin configuration. They were used for connecting keyboard, mice, and modems.

- **PARALLEL PORT:** This port is used to transfer data parallelly 8 bits at a time. This is implemented using 25 pins and usually for printers.

- **VGA (Video Graphics Array):** This 15-pin port allows transfer of analog video data to the monitor. This is replaced with DVI (Digital Video Interface) which uses digital data i.e. used by flat LCD panels.

- **RJ45 port:** This port is used by Ethernet network. Even though computer connects wirelessly, the wired RJ45 remains as a useful option.

- **PS/2 port:** This port has replaced serial port. It has 6 pins but occurs as a pair in two different colors. The ports and connectors for keyboard are purple, while the mouse uses green port. USB has invaded this area also.

- **HDMI (High Definition Multimedia Interface):** This is used for transferring audio and video between computers and HDTVs, projectors and home theaters.

9. INPUT DEVICES

Input devices are needed to interact with the OS to perform tasks. Ex: Keyboard, Mouse, Joystick, Stylus, Scanner etc.

- **Keyboard:** Every computer supports a keyboard – either a physical or touchscreen. The keyboard has QWERTY layout and contains large number of symbols. Each letter, numeral or symbol is known as a character, which represents smallest piece of information. Each character has a unique values called the ASCII (American Standard Code for Information Interchange) value.

- **Pointing Devices:** GUI (Graphical User Interfaces) like windows need a pointing device to control the movement of cursor on the screen. This is implemented as mouse in desktop and touchpad in laptop. The earliest form has a rotating ball and two buttons. The left button is used for selecting (by clicking once) and execute (by clicking twice). The right button is used to check and change attributes. The optical mouse uses infrared laser/LED, the wireless mouse uses radio frequency technology.

- **Scanner:** A scanner is a device that creates a digital image of a document by optically scanning it. The flatbed scanner doesn't exceed more than A4 size. It is operated with by using special software. The document to be scanned is placed on a glass plate covered by lid. Modern scanners have **OCR (Optical Character Recognition)** facility which extracts text as a stream of characters i.e. converting image file to text file and **MICR (Magnetic Ink Character Recognition)** reads the codes using hand held barcode readers.

10. OUTPUT DEVICES

The information produced can be heard or seen with the help of output devices. Some of the output devices are monitor, speaker, printer, plotter etc.

➢ **Monitor:** The monitor is an integral part of computer which displays both text and graphics. The performance is measured in terms of image quality, resolution, energy consumption.

CRT (Cathode Ray Tube) Monitors: The CRT monitors have a rarefied tube containing three electron guns. The guns emit electrons create images. They usually have resolution of 640*840 pixels. They are large, heavy, energy efficient and produces lot of heat.

LCD (Liquid Crystal Display) Monitors: The image is formed by applying voltage on crystals. The backlight is provided by fluorescent light. They consume less power, generate less heat and have increased life span.

➢ **Impact Printers:** It produces the hardcopy of output. The impact printers are old, noisy. But, still dot-matrix printer is in usage.

Dot-Matrix Printer: The print head of the dot-matrix printer has either 9 or 24 pins. When the pins fire against the ribbon, an impression is created on the paper. The speed is 300 cps and doesn't produce high quality output. It can produce multiple copies.

Daisy-wheel Printer: It employs a wheel with separate characters distributed along its outer edge. The wheels can be changed to obtain different set of fonts.

Line Printer: For heavy printing, the line printer is used. It uses a print chain containing the characters. Hammers strike the paper to print and it rotates at the speed of 1200 lpm. It is also noisy and is of low-quality.

➢ **Non-Impact Printers:** They are fast, quiet and of high resolution. The most commonly used non-impact printers are laser and ink-jet printers. The thermal printer is not discussed here because it uses heat to print on high-sensitive paper.

Laser Printer: It works like a photocopier and uses toner i.e. black magnetic powder. The image is created in the form of dots and passed from drums on to the paper. It has built-in RAM which acts as buffer and RAM to store fonts. The resolution varies from 300 dpi to 1200 dpi and the speed is about 20 ppm.

Ink-jet Printer: These are affordable printers. It sprays tiny drops of ink at high pressure as it moves along the paper. The separate cartridges are available for different colors. The resolution is about 300 dpi, can print 1 to 6 pages/min.

➢ **Plotters:** The plotter can make drawings. It uses one or more automated pens. The commands are taken from special file called vector graphic files. Depending on type of plotter either paper or pen moves. It can handle large paper sizes and it is used for creative drawings like buildings and machines. They are slow and expensive.

11. COMPUTERS IN A NETWORK

Most organizations, large or small, no longer have standalone computers. Computers cooperate with one another by being connected in a network. The hardware resources can also be shared. A printer can be used by several users if it is connected to a network.

Network Topology: The ways used to connect the computers. The different types are: Bus, Star, Ring, and Mesh.

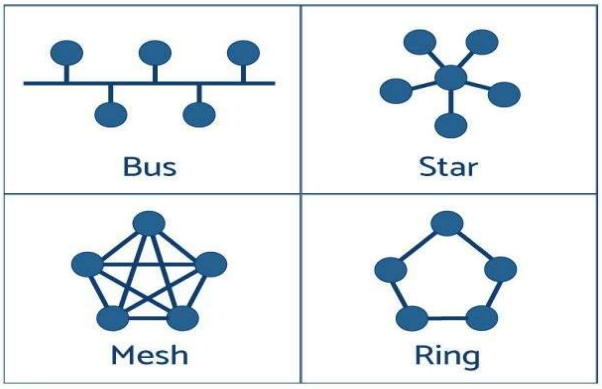


Figure 2: Network Topologies

1. **Bus Topology:** It uses a single cable called as bus to which all computers are connected. The failure of one single node doesn't disturb other nodes.
2. **Star Topology:** It uses a central hub to which all nodes are connected. If the hub fails, entire network fails and the nodes cannot be added easily.
3. **Ring Topology:** It is connected in the form of closed loop without hub. Data moves from node to node. For unidirectional rings, if one node fails, the network fails.
4. **Mesh Topology:** Nodes are connected to one another offering multiple paths. If node breaks down, then data passing changes its route. It is expensive.

- **Network Types:** Networks are also classified on their size. The most common types are LAN and WAN. And some other types are: MAN, CAN and PAN.

1. **LAN (Local Area Network):** They are used in smaller organizations usually using Ethernet. The usual speed will be 100Mbps.
2. **WAN (Wide Area Network):** They are used in cities and can connect larger distance. They use optic fiber cables. Banks, Airline and Hotel reservations use WANs.
3. **MAN (Metropolitan Area Network):** It is sandwiched between LAN and WAN and used for interconnecting in same cities.
4. **PAN (Personal Area Network):** This is the smallest network and can connect only few meters. It connects small devices like phones, laptops through Bluetooth.

- ***Intranet*** is a network of computers designed for a specific group of users and can be accessed from Internet but with restrictions.
- ***Internet*** is wide network of computers and is open for all and itself contains a large number of intranets.

12. NETWORK HARDWARE

Connecting computers in a network require additional devices that are not part of the computer's basic configuration.

- **Hub and Switch:** Computers in a single network is connected to a central device called hub or switch. ***HUB*** accepts network data from computer and broadcasts to the nodes by checking destination address.

SWITCH will have a table which contains MAC addresses of connected devices. The data is sent after looking up the table for destination.

- **Bridge and Router:** The network supports many nodes which leads to congestion. Hence, the network may be split into a number of segments, with a ***BRIDGE*** connecting them. It connects two networks using the same protocol. ***ROUTER*** connects two similar or dissimilar networks separated by long distance. It is a part of two networks and thus have two addresses. It has a routing table to store address.

13. SOFTWARE BASICS

- Software is a collection of code that drives a computer to perform related group of tasks.
- Programs in software use a language.
- The source code is created by programmer using programming languages like C, C++, Java, Python etc.
- The software is developed to operate on multiple platforms.

SOFTWARE TYPES

Computer software can be broadly classified into two types: System software and Application Software.

- **System Software:** Software run by the computer to manage the hardware connected to it is known as system software. System software examples:
  BIOS- It checks the hardware devices and peripherals at boot time.
  OS: It manages both hardware and programs running on the computer.

- **Application Software:** Software not directly connected with hardware but related to a specific application is known as application software. Application software examples:
  Office software: This comprises of three separate applications: word processing, spread sheet and presentations.
  Database software: It allows data to have a uniform structure to be stored in a database.

System Software	Application Software
System software is used for operating computer hardware.	Application software is used by user to perform specific task.
System softwares are installed on the computer when operating system is installed.	Application softwares are installed according to user's requirements.
In general, the user does not interact with system software because it works in the background.	In general, the user interacts with application softwares.
System software can run independently. It provides platform for running application softwares.	Application software can't run independently. They can't run without the presence of system software.
Some examples of system softwares are compiler, assembler, debugger, driver, etc.	Some examples of application softwares are word processor, web browser, media player, etc.

CHAPTER 2: INTRODUCTION TO C

1. PSEUDOCODE

- Pseudocode is an informal list of steps in English like statements which is intended for human reading.
- It is used in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

Advantages:

1. It can be written easily
2. It can be read and understood easily
3. Modification is easy

Disadvantages:

1. There is no standard form.
2. One Pseudocode may vary from other for same problem statement.

Examples:

Pseudocode: Addition of two numbers	**Pseudocode:** Area of rectangle
1. Begin 2. Read two numbers a, b 3. Calculate sum = a+b 4. Display sum 5. End	1. Begin 2. Read the values of length, breadth 3. Calculate area = l x b 4. Display area 5. End

2. HISTORY OF C

- C is a general purpose, procedural, structured computer programming language developed by **Dennis Ritchie** in the year 1972 at AT&T Bell Labs.
- C language was developed on UNIX and was invented to write UNIX system software.
- C is a successor of B language.
- There are different C standards: K&R C std, ANSI C, ISO C.

Characteristics of C:

- C is easy to learn.
- C is a general purpose language.
- C is a structured and procedural language.
- It is portable.
- It can extend itself.

Examples of C:

- Operating system
- Language compilers
- Assemblers

- Text editors
- Databases

3. THE BASIC 'C' CHARACTERS

- A C character set defines the valid characters that can be used in a source program. The basic C character set are:

1. **Letters:** Uppercase: A,B,C,….,Z
   Lowercase: a,b,c,…..,z
2. **Digits:** 0,1,2,…..,9
3. **Special characters:** ! , . # $ ( ) } { etc
4. **White spaces:** Blank space, Horizontal tab space (\t), carriage return, new line character (\n), form feed character.

4. C – TOKENS

- It is also called as *lexical unit.*
- The smallest individual unit in 'C' program is called as C-Token.
- These are the building blocks of 'C' program.
- There are 6 types of 'C' tokens:
  i. Keywords
  ii. Identifiers
  iii. Constants
  iv. Strings
  v. Operators
  vi. Special symbols

i. Keywords

- Keywords are also called as *reserved words.*
- They already have pre-defined meaning.
- Keywords should always be written in lowercase.
- The keywords meaning can't be changed.
- Keywords cannot be used as identifiers.
- There are 32 keywords in 'C' program.

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

ii. Identifiers

- It refers to the name of the objects.
- These are the names given to variables, functions, macros etc.
- The rules to write an identifier are as follows:
  1. It must contain only alphabets (A to Z, a to z), numbers (0 to 9) and underscore (_).
  2. It must start with alphabet or underscore but not numeric character.
  3. It should not contain any special symbols apart from underscore.
  4. It should not have any space.
  5. Keywords cannot be used as identifiers.
  6. Identifiers are case sensitive.
  7. Maximum length of an identifier is 31 characters.
- ***Examples:***

Valid Identifiers: integer, minimum, sum_total, row1, _cpps

Invalid Identifiers: float → It is a keyword.

I am → It has space

123_Abc → It is starting with number

N1 + n2 → It contains special symbol (+)

iii. Constants

Constants are the values that doesn't changes during the execution of a program.

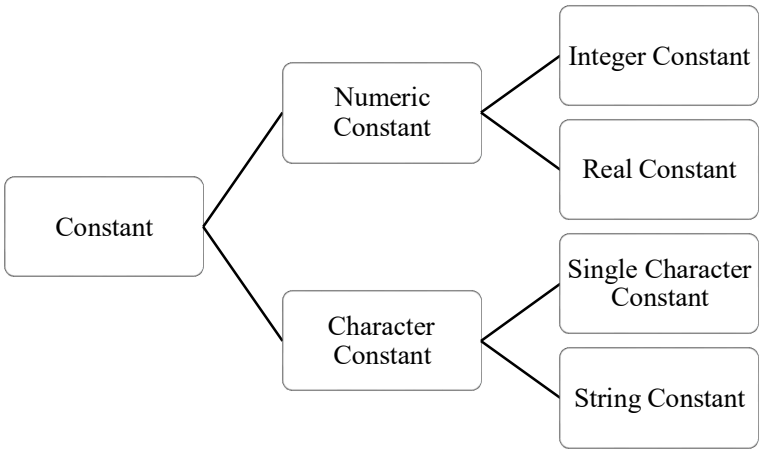


Figure 3: Types of Constants

✓ **Numerical Constant**

- **Integer Constant:** These are the numbers without decimal part.
  1. **Decimal:** It is an integer constant consisting of numbers from 0-9. It can be preceded by + or –
  2. **Octal:** It is an integer constant consisting of numbers from 0-7. It is preceded by o

3. **Hexadecimal:** It is an integer constant consisting of numbers from 0-9, A-F (A=10, B=11, C=12, D=13, E=14, F=15). It is preceded by 0x

➢ **Real Constant:**

- These are the numbers with decimal part.
- These are also called as floating point constant.
- The floating values can be identified by the suffix 'F' or 'f'
- It can be expressed in exponential form.
- Ex: 21.5, 2.15 x 10^2 → 2.15e2

✓ **Character Constant**

➢ **Single Character Constant:**

- These are the values enclosed within single quotes.
- Each character have an integer value called as ASCII (American Standard Code for Information Interchange) → A: 65, a: 97
- *Printable Character:* All characters are printable except backslash(\) character or escape sequence.
- *Non-Printable Character:* The characters with backslash (\) are non printable. Ex: \n, \t

➢ **String Constant:**

- A group of characters together form string.
- Strings are enclosed within double quotes.
- They end with NULL character (\0).
- Ex: "CBIT" →

C	B	I	T	\0

NOTE:

✓ **Qualified Constant/ Memory Constant:** These are created by using "const" qualifier. Const means that something can only be read but not modified.

Syntax: const datatype variable_name = value;

Ex: const int a = 10;

const float pi = 3.14;

✓ **Symbolic Constants/ Defined Constant:** These are created with the help of define preprocessor directive.

Ex: #define PT 3.142

- During processing the PI in the program will be replaced with 3.142.
- The name of constant is written in uppercase just to differentiate from the variables.

✓ **Enumeration Constant:** An enumeration is a list of constant integer values. Names in enumeration must be distinct. The keyword used is "enum".

Ex: enum days = {Sun, Mon, Tue, Wed, Thu, Fri, Sat}

Sun → 1, Mon → 2, ……., Sat → 7

iv. Strings

- A group of characters together form string.
- Strings are enclosed within double quotes.
- They end with NULL character (\0).
- Ex: "CPPS" →

C	P	P	S	\0

V. Operators

An operator is a symbol that tells the compiler to perform specific mathematical and logical functions.

The different operators supported in 'C' are:

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

1. Arithmetic Operators: The different arithmetic operators are:

Operator	Name	Result	Syntax	Example (b=5, c=2)
+	Addition	Sum	a = b + c	a = 7
-	Subtraction	Difference	a = b - c	a = 3
*	Multiplication	Product	a = b * c	a = 10
/	Division	Quotient	a = b / c	a = 2
%	Modulus	Remainder	a = b % c	a = 1

2. **Relational Operators:** These are used to compare two quantities. The output will be either 0 (False) or 1 (True). The different relational operators are:

Operator	Name	Syntax	Example (b=5, c=2)
<	Lesser than	a = b < c	a = 0 (False)
>	Greater than	a = b > c	a = 1 (True)
<=	Lesser than or Equal to	a = b <= c	a = 0 (False)
>=	Greater than or Equal to	a = b >= c	a = 1 (True)
==	Equal to	a = b == c	a = 0 (False)
!=	Not equal to	a = b != c	a = 1 (True)

3. **Logical Operators:** These are used to test more than one condition and make decision. The different logical operators are: NOT, AND, OR

   ✓ **Logical NOT (!)** The *output is true* when *input is false* and vice versa. It accepts only one input.

Input	Output
X	**!X**
0	1
1	0

   ✓ **Logical AND (&&)** The *output is true* only if *both inputs are true*. It accepts two or more inputs.

Input		Output
X	**Y**	**X && Y**
0	0	0
0	1	0
1	0	0
1	1	1

   ✓ **Logical OR (||)** The *output is true* only if *any of its input is true*. It accepts two or more inputs.

Input		Output
X	**Y**	**X || Y**
0	0	0
0	1	1
1	0	1
1	1	1

4. **Assignment Operators:** These are used to assign the result or values to a variable. The different types of assignment operators are:

Simple Assignment	a = 10
Shorthand Assignment	a += 10 → a = a + 10
Multiple Assignment	a = b = c = 10

5. **Bitwise Operators:** These works on bits and performs bit by bit operations. The different types of bitwise operators are:
   i. Bitwise NOT (~)
   ii. Bitwise AND (&)
   iii. Bitwise OR (|)
   iv. Bitwise XOR (^) → Output is True when odd number of 1's are present.
   v. Bitwise left shift (<<)
   vi. Bitwise right shift (>>)

✓ **Bitwise NOT (~)**

X	~X
0	1
1	0

✓ **Bitwise AND (&), Bitwise OR (|),Bitwise XOR (^)**

X	Y	X & Y	X | Y	X ^ Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

✓ **Bitwise Left Shift (<<)** → Shift specified number of bits to left side.

X	0	1	0	0	0	1	1	0
X<<2	0	0	0	1	1	0	0	0

✓ **Bitwise Right Shift (>>)** → Shift specified number of bits to right side.

X	0	1	0	0	0	1	1	0
X>>2	0	0	0	1	0	0	0	1

6. **Unary Operators:** There are 4 types:

✓ Unary Plus Operator
✓ Unary Minus Operator
} → Determines Sign
✓ Increment (++)
✓ Decrement (--)

✓ **Increment (++):** It adds one to the operand.

Pre-increment	Post-increment
First value of the operand is incremented (added) by 1 then, it is used for evaluation.	First value of the operand is used for evaluation then, it is incremented (added) by 1.
Ex: ++a	**Ex:** a++

✓ **Decrement (--):** It subtracts one from the operand.

Pre-decrement	Post- decrement
First value of the operand is decremented (subtracted) by 1 then, it is used for evaluation.	First value of the operand is used for evaluation then, it is decremented (subtracted) by 1.
Ex: - -a	**Ex:** a- -

7. **Conditional Operator/ Ternary Operator (?:)**

✓ It takes three arguments.

Expression1 ? Expression2 : Expresseion3

Where,
Expression1 → Condition
Expression2 → Statement followed if condition is true
Expression3 → Statement followed if condition is false

✓ Ex: large = (4 > 2) ? 4: 2 → large = 4

8. **Special Operator/ Special Symbols**

✓ **Comma Operator:** It can be used as operator in expression and as separator in declaring variables.
✓ **Sizeof Operator:** It is used to determine the size of variable or value in bytes.
✓ **Address Operator:** It is used to find the address of the operators.

5. EXPRESSIONS

✓ It is combination of operands (variables, constants) and operators.
✓ ***Precedence:*** The order in which operators are evaluated is based on the priority value.
✓ ***Associativity:*** It is the parsing direction used to evaluate an expression. It can be left to right or right to left.
✓ ***Evaluation of expressions:*** Expressions are evaluated using an assignment statement.
✓ ***Ex:*** variable = expression

sum = a + b

✓ Following table provides the Precedence and Associativity of operators:

Operator	Description	Associativity	Precedence(Rank)
( ) [ ]	Function call Array element reference	Left to right	1
+ - ++ -- ! ~ * & Sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer to reference Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
<< >>	Left shift Right Shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13

= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

6. DATA TYPES

- ✓ Data type defines the types of data that is stored in a variable.
- ✓ There are 3 types:
  - i. Primary/ Built-in/ Fundamental data type → int, float, double, char, void
  - ii. Derived data type → array, structure
  - iii. User defines data type → enum, typedef

❖ **Primary/ Built-in/ Fundamental data type**: These are the built in data types available in C. There are 5 types. Namely:

1. Integer (int)
2. Floating point (float)
3. Character (char)
4. Double precision floating point (double)
5. Void (void)

1. Integer type:

- ➢ It is used to store whole numbers.
- ➢ The keyword int is used to declare variable of integer type.
- ➢ Int type takes 2B or 4B of storage depending on machine size.
- ➢ The classes of integer are:

Unsigned			Signed		
Data type	**Keyword**	**Size**	**Data type**	**Keyword**	**Size**
Short Integer	short int	1B	Signed short integer	signed short int	1B
Integer	int	2B	Signed integer	signed int	2B
Long Integer	long int	4B	Long integer	long int	4B

2. Floating point data type:

- ➢ It is used to store decimal numbers.
- ➢ The keyword float is used to declare variable of floating point data type.

- Float type takes 4B or 8B of storage depending on machine size.
- It can be expressed in fractional or exponential form.

Data type	Keyword	Size
Floating point	float	4B
Double	double	8B
Long double	long double	10B

3. Double:

- It is used to store double precision floating point numbers.
- The keyword double is used to declare variable of floating point data type.
- Double type takes 8B of storage.
- Double precision is related to accuracy of data.

4. Char:

- It is used to store character type of data.
- The keyword char is used to declare variable of character type.
- Char type takes 1B of storage.

5. Void:

- It is a special data type that has no value.
- It doesn't have size.
- It is used for returning the result of function that returns nothing.

7. VARIABLES

✓ A variable is a name given to memory location whose value changes during execution.

✓ ***Declaration:***

Syntax: datatype variable_list;

Where,

datatype → Any built in data type

variable_list → Identifiers that specifies variable name.

Example: int a;

Where,

int is the built in data type

a is variable of type int.

✓ ***Initialization:***

Syntax: datatype variable_name = value;

Where,

datatype → Any built in data type
variable_name → Identifier that specifies variable name.
value → The data which will be stored in variable name.

Example: int a = 10;

Where,

int → data type
a → variable name
10 → value

8. TYPE CONVERSION

- ✓ Converting the value of one data type to another type is called as Type Conversion.
- ✓ It occurs when mixed data occurs.
- ✓ There are two types:

i. Automatic Type Conversion (Implicit)

- ✓ Here, the operand/ variables of smaller data type is automatically converted to data type of larger size.
- ✓ char → int → long int → float → double → long double
- ✓ **Ex:** int a = 5;
  float b = 25, c;
  c = a / b;
  printf("%f" , c);

a	b	c
5	25.0	0.25

ii. Manual Type Conversion (Explicit)

- ✓ It is a forced conversion used to convert operand/ variables of larger data type to smaller size or vice versa.
- ✓ **Ex:** int a = 7, c;
  float b = 4.0;
  b = a % (int) b;
  printf("%d" , c);

a	b	c
7	4.0	3
b → Converted into int & evaluated		

9. STRUCTURE OF 'C' PROGRAM

- ➢ C program consists of different sections. The structure of C program is as shown below.
- ➢ **Documentation Section:** It consists of a set of comment line giving the name of the program, the author and other details. Compiler ignores these comments. There are two formats:
  - **a)** Line comments //Single line comment
  - **b)** Block comments /*Multi line comment*/

- **Link Section:** It provides instruction to the compiler to link functions from system library.
- **Definition Section:** It defines all symbolic constants.
- **Global Declaration Section:** The variables which are used in more than one function are called global variables and are declared in this section.
- **main( ) Section:** Every 'C' program has one main( ) function and it contains two parts:
  i. **Declaration Part:** Here, all the variables used are declared.
  ii. **Executable Part:** Here, it contains at least one executable statement.

  The main( ) function is enclosed within flower brackets and the statements ends with semicolon.
- **Sub-program Section:** It contains all the user defined functions that are called by main( ) function.

```
Documentation section
Link  section
Definition section
Global declaration section
main( ) Section
{
        Declaration part
        Executable part
}
Sub program section
Function 1
Function 2
        -
        -
Function n
```

- **Example:**

```
//Addition of two numbers
#include<stdio.h>
void main( )
{
        int a, b, sum;
        printf("Enter two numbers\n");
        scanf("%d%d", &a, &b);
        sum = a + b;
        printf("The sum is %d", sum);
}
```

Questions:

1. Evaluate each of the following expression independent of each other.
   i. x = a – b/3 + c * 2 – 1 where, a=9, b=12, c=3
   ii. 10 != 10 || 5 < 4 && 8
   iii. 100%20 <= 20 – 5 + 100 % 10 – 20 = = 5 >=1 != 20
   iv. a += b * = c - = 5 where, a = 3, b=5 and c=8
   v. int i=3, j=4, k=2
      a) i++ - j
      b) ++k % --j
      c) j+1/i-1
      d) j++/i--
   vi. 22+3 < 6 && !5 || 22 == 7 && 22-2 > +5
   vii. A+2 b || !c && a ==d*a-2<=e where, a=11, b=6, c=0, d=7 and e=5

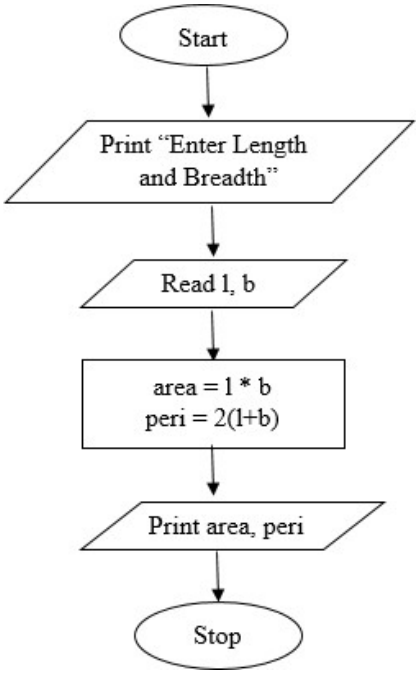
2. Classify the following into input and output devices:
   Monitors, Visual display unit, Track balls, Barcode reader, Digital camera, Film recorder, Microfiche, OMR, Electronic Whiteboard, Plotters.

3. State whether the following are valid identifiers or not:
   integer, float, Iam, 123_ABC

4. Convert the following mathematical expressions into 'C' expressions:
   i) X = e power root of(x) + e power root of(y)/xsin root(y)
   ii) C = a*a+1/((b+1)/(c+d))
   iii) C = 3 root of((a*a*a+b*b*b*b)/(a*a*a*a-b*b*b))
   iv) Area = ((pie*r*r) + (2*pie*r*h))
   v) Area = $\boldsymbol{\pi r^2 + 2\,\pi\, rh}$
   vi) x = $\frac{-b+\sqrt{b^2-4ac}}{2a}$
   vii) m = $x^4 + \sqrt{x + \frac{y}{k}} - 4x + 6$

Example Programs:

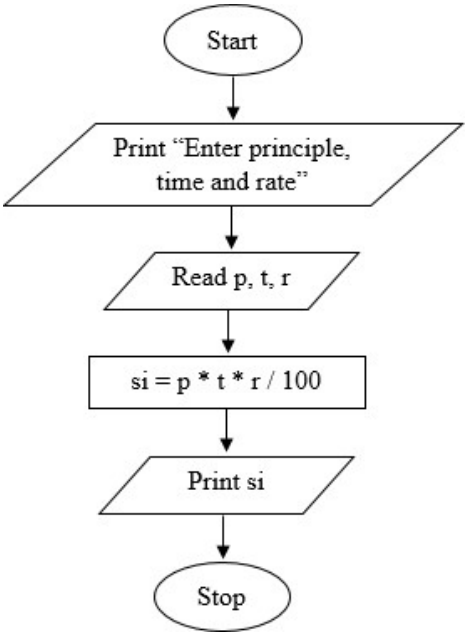
Program 1: To find area and perimeter of rectangle

Algorithm	Flowchart
Step 1. [Initialize] Start Step 2. [Input the length and breadth] Read l, b Step 3. [Calculate area] area = l * b Step 4. [Calculate perimeter] peri= 2(l+b) Step 5. [Display area and perimeter] Print area, peri Step 6. [Finished] Stop.	

Program	Output

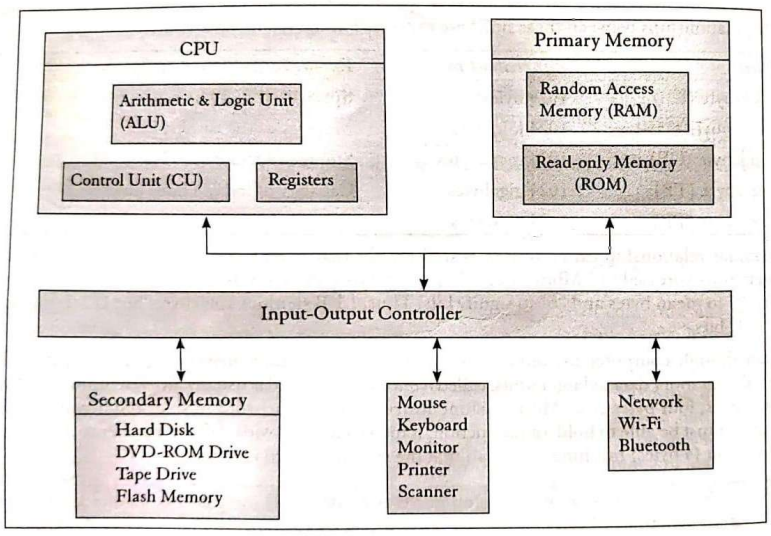
#include<stdio.h>
{
int l, b, area, peri;
printf("Enter length and breadth\n");
scanf("%d%d", &l, &b);
area = l * b;
peri = 2 * (l + b);
printf("The area is %d\n", area);
printf("The perimeter is %d\n", peri);
}
	Enter length and breadth
	2
	4
	The area is 8
	The perimeter is 12

Program 2: To find Simple Interest

Algorithm	Flowchart
Step 1. [Initialize] Start Step 2. [Input the principle, time and rate] Read p, t, r Step 3. [Calculate si] si = p * t * r / 100 Step 4. [Display si] Print si Step 5. [Finished] Stop.	
Program	**Output**

#include<stdio.h>
{
float p, t, r, si;
printf("Enter principle, time and rate\n");
scanf("%f%f%f", &p,&t,&r);
si = p * t * r / 100;
printf("The Simple Interest is %f", si);
}
	Enter principle, rate and time
	1000
	2
	4
	The Simple Interest is 80

Figure: Architecture of a Computer with a Single Processor



MODULE-2

CHAPTER 1: MANAGING INPUT AND OUTPUT OPERATIONS

1. INTRODUCTION

- ✓ Reading, processing and writing of data are three essential functions of a computer program.
- ✓ There are two methods. The first method is to assign the values and the second method is to use input function scanf to read data and output function printf to write data.
- ✓ Ex: **#include<stdio.h>** Includes function calls such as printf() and scanf() into the source code when compiled.

2. READING A CHARACTER

- ✓ The simplest way of all input/output operations is reading a character from the 'standard input' unit (usually keyboard) and writing it to the 'standard output' (usually the screen).
- ✓ Reading a single character can be done by using the function getchar and it does not have any parameter.

Syntax: variable_name = getchar( );

Where, variable_name is valid and has been declared in char type.

Example: char name;
name = getchar( );

- ✓ The getchar function can also be used to read the characters successively until the return key is pressed.
- ✓ The character function are contained in the file type ctype.h and therefore the preprocessor directive #include<ctype.h> is included.
- ✓ And the functions of ctype.h returns 1 if (TRUE) and 0 if (FALSE).

Function	Test
isalnum(c)	Is c alphanumeric character?
isalpha(c)	Is c alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c a lower case letter?
isupper(c)	Is c a upper case letter?
isspace(c)	Is c a white case character?
ispunct(c)	Is c a punctuation mark?
isprint(c)	Is c a printable character?

Example Program:

```
#include<stdio.h>

#include<ctype.h>
```

```
void main( )
{
        char key;
        printf("Enter any key\n");
        key=getchar( );
        if(isalpha(key))
                printf("The entered key is a letter\n");
        else if(isdigit(key))
                printf("The entered key is a digit\n");
        else
                printf("The entered key is not alphanumeric\n");
}
```

OUTPUT:

Enter any key A The entered key is a letter	Enter any key 7 The entered key is a digit	Enter any key $ The entered key is not alphanumeric

3. WRITING A CHARACTER

✓ Writing a single character can be done by using the function putchar.

Syntax: putchar(variable_name);

Where, variable_name is valid and has been declared in char type.

Example:
```
name='y';
putchar(name);
```
Will display the character y on the screen

✓ The characters entered can be converted into reverse i.e., from upper to lower case and vice versa.
✓ The functions used to convert are toupper, tolower.

Example Program:

```
#include<stdio.h>
#include<ctype.h>
void main( )
{
        char alphabet;
        printf("Enter any alphabet\n");
```

```
    alphabet=getchar( );
    if(islower(alphabet))
        putchar(toupper(alphabet));
    else
        putchar(tolower(alphabet));
}
```

OUTPUT:

Enter any alphabet A A	Enter any alphabet a A

4. FORMATTED INPUT

- ✓ Formatted input refers to an input data that has been arranged in particular format.
- ✓ Ex: 152 Rajesh
- ✓ In the above example the first part contains integer and the second part contains characters. This is possible with the help of scanf function.

Syntax: scanf("control string", arg1, arg2,…..,argn);

- ✓ The control string specifies the field form in which data should be entered and the arguments arg1, arg2,…..,argn specifies the address of the location where the data is stored.
- ✓ The width of the integer number can be specified using %wd.

Example-2:
- scanf("%2d %2d", &a, &b);
%2d→ two digit integer
%2d → two digit integer
4567 25
a=45 b=67
Note: 25 is not at all stored

- ✓ The input field may be skipped using * in the place of field width.

Example-3:
- scanf("%2d %*d %2d", &a, &b);
%2d→ two digit integer
%*d → skip this integer
45 25 63
a=45
25 is skipped
b=63

- ✓ The scanf function can read single character or more than one character using %c or %s.

Rules for scanf:

- ✓ Each variable to be read must have a specification
- ✓ For each field specification, there must be a address of proper type.
- ✓ Never end the format string with whitespace. It is a fatal error.
- ✓ The scanf will read until:
  - A whitespace character is found in numeric specification.
  - The maximum number of characters have been read
  - An error is detected
  - The end of file is reached
- ✓ Any non-whitespace character used in the format string must have a matching character in the user input.

❖ The below table shows the scanf format codes commonly used.

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%e, %f, %g	Read a floating point value
%h	Read a short integer
%i	Read a decimal integer
%o	Read a octal integer
%s	Read a string
%u	Read an unsigned decimal integer
%x	Read a hexadecimal integer
%[ ]	Read a string of word

The l can be used as a prefix for long integer. Ex: %ld

5. FORMATTED OUTPUT

✓ The printf function is used for printing the captions and results.

Syntax: printf("control string", arg1, arg2, ….., argn);

✓ Control string contains three types of items:

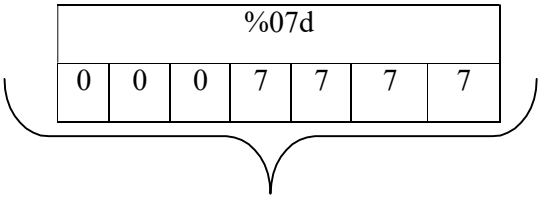
1. Characters that will be printed on the screen as they appear.
2. Format specifications that define the output format for display of each item.
3. Escape sequence characters such as \n, \t and \b.

✓ The printf can contain flags (0 + -)

0 → fill in extra positions with 0's instead of spaces

a=7777

printf("a = %07d", a);

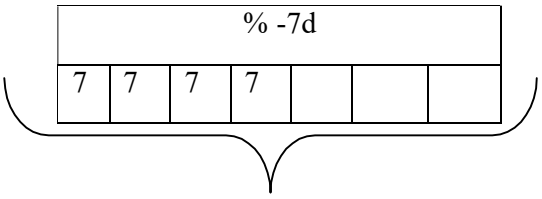


Width of 7 memory space is allocated and padded with leading zeroes

- → Left justify text instead of right justify text

a=7777;

printf("a = % -7d", a);



Width of 7 memory space is allocated and printed from left side (left justify)

+→Print sign of number (whether positive or negative)

- Example with format specifier %d:

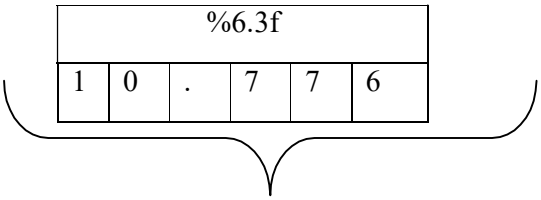
a=10, b=77;
printf("%d %d", a , b);

Output:

10 77

- Example with format specifier %f:

x=10.776;
printf("%6.3f %", x);



6.3f means totally 6 locations are allocated out of which 3 is for number of decimal places.

- Example with \n and \t:

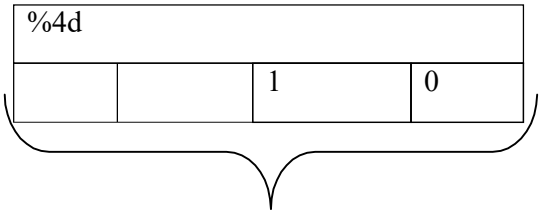
a=10, b=77, c=88;
printf("a=%d\n b=%d\tc=%d", a , b, c);

Output:

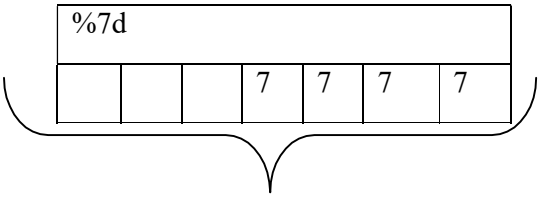
a=10
b=77
c=88

- Example with size value:

a=10, b=7777, c=88;
printf("a=%4d\n b=%7d", a , b);



Width of 4 memory space is allocated and printed from right side



Width of 7 memory space is allocated and printed from right side (right justify)

❖ The below table shows the scanf format codes commonly used.

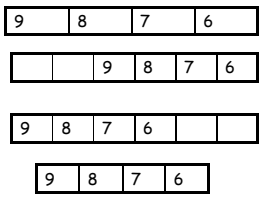
Code	Meaning
%c	Print a single character
%d	Print a decimal integer
%e	Print a floating point value in exponent form
%f	Print a floating point value without exponent
%g	Print a floating point value with or without exponent
%h	Print a short integer
%i	Print a decimal integer
%o	Print a octal integer
%s	Print a string
%u	Print an unsigned decimal integer
%x	Print a hexadecimal integer
%[ ]	Print a string of word

The l can be used as a prefix for long integer. Ex: %ld

➢ General syntax of field-width/size for different data types:

- %wd → Example: %6d
- %w.pf → Example %6.2f
- %w.ps → Example %10.4s
  - w means total field size/width
  - P means precision (tells how many decimal places to be printed in case of float) and
  - How many characters to be printed in case of strings

```
Examples:
a=9876;
printf("%d", a);
printf("%6d",a);
printf("%-6d",a);
printf("%3d",a);
```



NOTE: String input/output function: gets() and puts( )

➢ gets and puts are line oriented input output functions available in the <stdio.h> header file.

➢ **gets():**Alternative method to read strings is ***gets() (read it as get string)*** function which is an unformatted string input function. Advantage of ***gets()***is it can read multi-word strings.

➢ Example:char name[10];
gets(name);

Say input is:

N	E	W		Y	O	R	K

The complete string "NEW YORK" is read by ***gets( ).***

➢ **puts():**This function can be used to display entire string on monitor screen without the help of **%s** specifier.

➢ Example: char name[ ] = "Bengaluru"

puts(name);

puts( ) will display the string value stored in parameter passed to it on monitor. In this case it is—Bengaluru.

CHAPTER 2

Decision Making and Branching

1. INTRODUCTION

✓ C language possesses such decision making capabilities by supporting the following statements:
1. **If** statement
2. **Switch**statement
3. Conditional operator statement
4. **goto** statement

✓ These statements are popularly known as decision making statements. Also known as control statements.

2. DECISION MAKING (Two-Way Selection Statements)

✓ The basic decision statement in the computer is the two way selection.
✓ The decision is described to the computer as conditional statement that can be answered TRUE or FALSE.
✓ If the answer is TRUE, one or more action statements are executed.
✓ If answer is FALSE, the different action or set of actions are executed.
✓ Regardless of which set of actions is executed, the program continues with next statement.
✓ C language provides following two-way selection statements:

1. if statement
2. if – else statement
3. Nested if else statement
4. Cascaded if else (also called else-if ladder)

1. if statement: The general form of simple if statements is shown below.

```
if (Expression)
{
   Statement1;
 }
Statement2;
```

- ✓ The Expression is evaluated first, if the value of Expression is true (or non zero) then Statement1 will be executed; otherwise if it is false (or zero), then Statement1 will be skipped and the execution will jump to the Statement2.
- ✓ Remember when condition is true, both the Statement1 and Statement2 are executed in sequence. This is illustrated in Figure1.

Note: Statement1 can be single statement or group of statements.

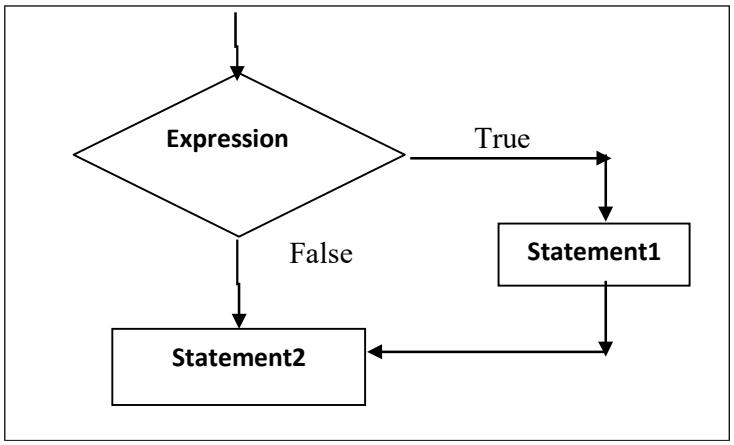


Figure 1: Flow chart of if statement

Example:

```
#include<stdio.h>
void main( )
{
        int a=20, b=11;
        if (a >b)
        {
                printf("A is greater\n");
        }
}
```

Output: A is greater

2. **if..else statement:** The if..else statement is an extension of simple if statement.

```
if (Expression)
{
    Statement1; → true-block
}
else
{
   Statement2; →true-block
}
  Statement3;
```

- ✓ If the Expression is true (or non-zero) then Statement1 will be executed; otherwise if it is false (or zero), then Statement2 will be executed.

- ✓ In this case either true block or false block will be executed, but not both.
- ✓ This is illustrated in Figure 2. In both the cases, the control is transferred subsequently to the Statement3.

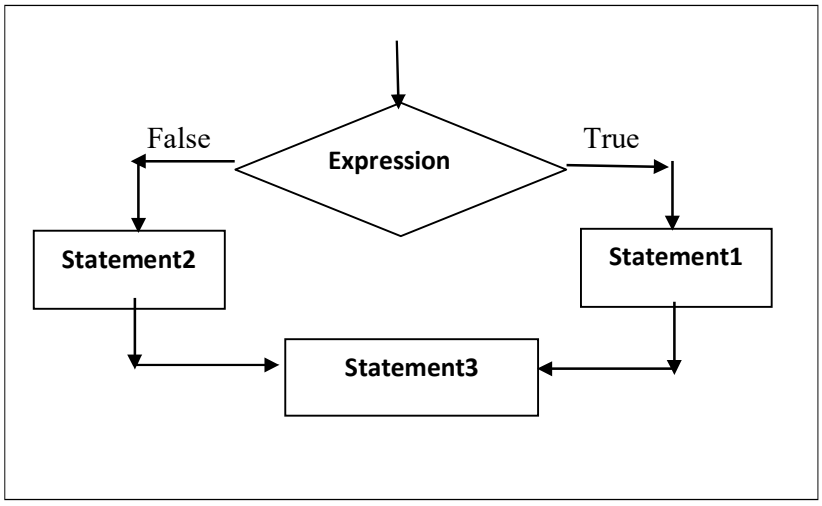


Figure 2: Flow chart of if-else statement

Example:

```
void main( )
{
        int a=10, b=11;
        if (a >b)
        {
                printf("A is greater\n");
        }
        else
        {
                printf("B is greater\n");
        }
}
```

Output: B is greater

3. Nested if .. else statement: When a series of decisions are involved, we have to use more than one if..else statement in nested form as shown below in the general syntax.

```
if (Expression1)
{
  if(Expression2)
   {
        Statement1;
   }
  else
   {
        Statement2;
    }
}
```

```
else if(Expression3)
{
        Statement3;
}
else
{
        Statement4;
}
```

- ✓ If Expression1 is true, check for Expression2, if it is also true then Statement1 is executed.
- ✓ If Expression1 is true, check for Expression2, if it is false then Statement2 is executed.
- ✓ If Expression1 is false, then Statement3 is executed.
- ✓ Once we start nesting if .. else statements, we may encounter a classic problem known as dangling else.
- ✓ This problem is created when no matching else for every if.
- ✓ C solution to this problem is a simple rule "always pair an else to most recent unpaired if in the current block".
- ✓ Solution to the dangling else problem, a compound statement.
- ✓ In compound statement, we simply enclose true actions in braces to make the second if a compound statement.

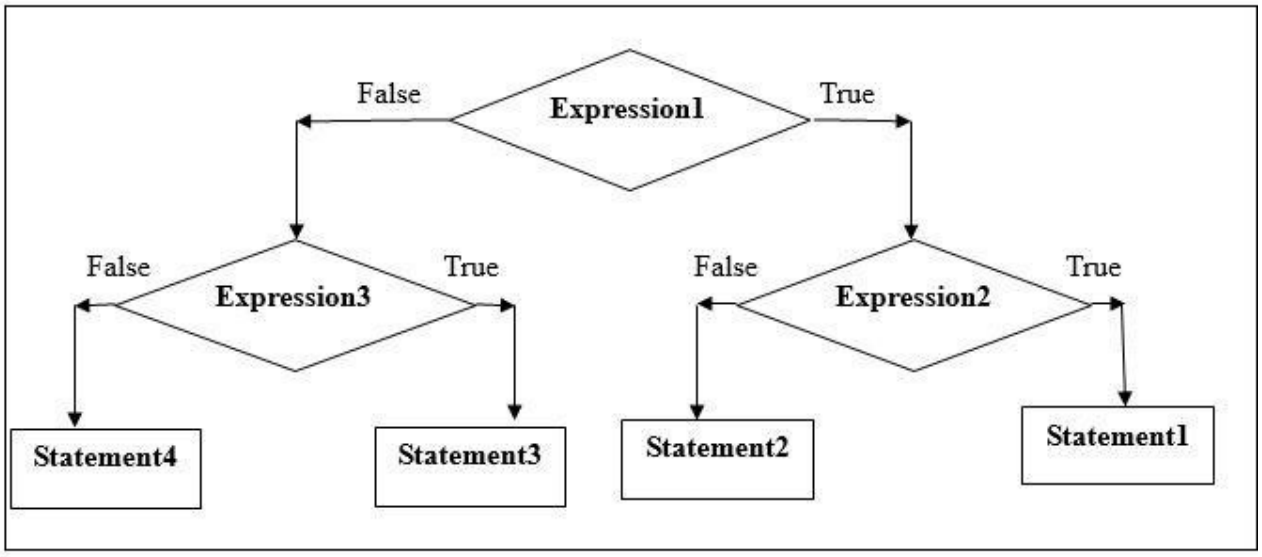


Figure 3: Flow chart of Nested if-else statement

Example:

```
#include<stdio.h>
void main( )
{
        int a = 20, b=15, c=3;
        if(a>b)
        {
                if(a>c)
                {
                        printf("A is greater\n");
                }
                else
```

```
        {
                printf("C is greater\n");
        }
}
else
{
        if(b>c)
        {
                printf("B is greater\n");
        }
        else
        {
                printf("C is greater\n");
        }
}
```

Output: A is greater

4. else-if ladder or cascaded if else: There is another way of putting ifs together when multipath decisions are involved. A multi path decision is a chain of ifs in which the statement associated with each else is an if. It takes the following form.

```
if (Expression1)
{
        Statement1;
}
        else if(Expression2)
        {
                Statement2;
        }
                else if(Expression3)
                {
                        Statement3;
                }
                        else
                        {
                                Statement4;
                        }
Next Statement;
```

- ✓ This construct is known as the else if ladder.
- ✓ The conditions are evaluated from the top (of the ladder), downwards. As soon as true condition is found, the statement associated with it is executed and control transferred to the Next statement skipping the rest of the ladder.
- ✓ When all conditions are false then the final else containing the default Statement4 will be executed.

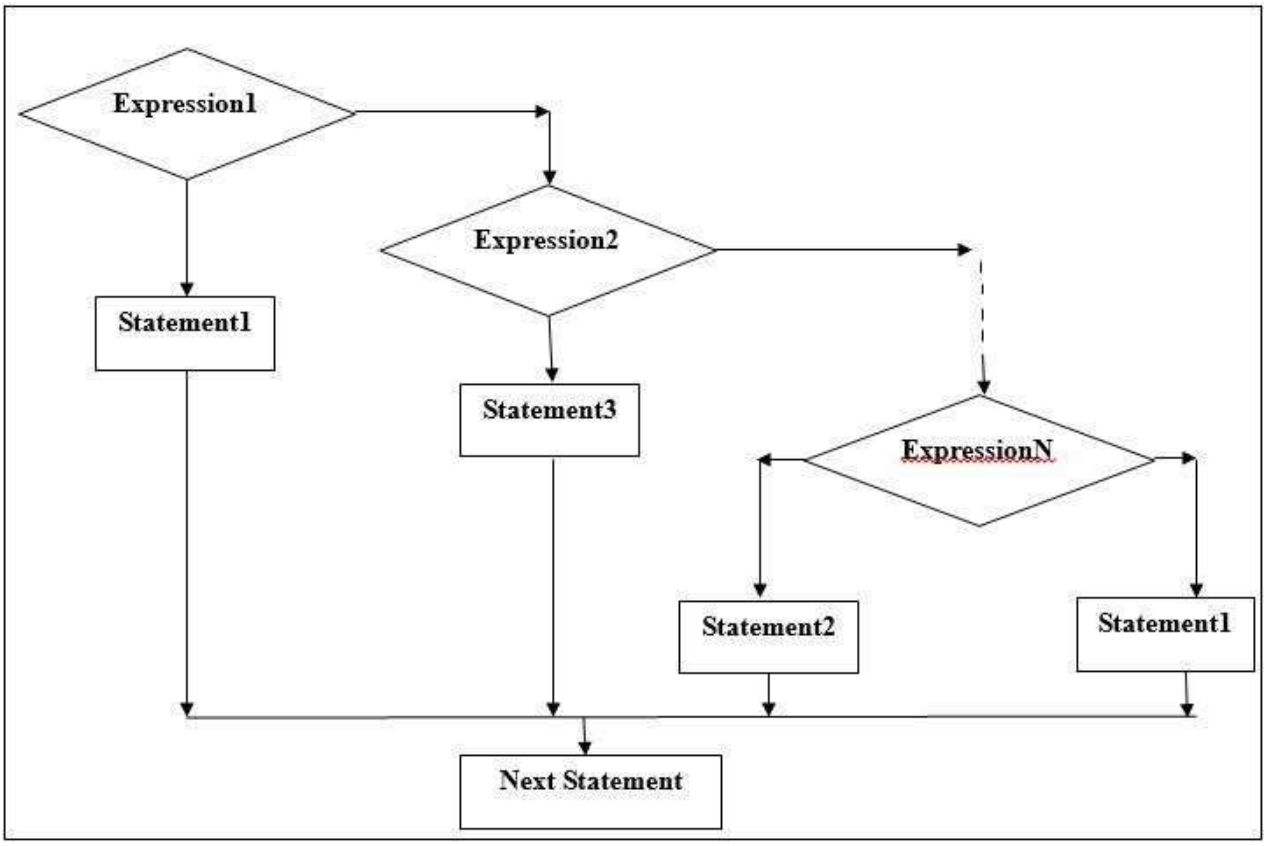


Figure 4: Flow chart of else if ladder statement

Example:

```
#include<stdio.h>
void main( )
{
        int a=20, b=5, c=3;
        if((a>b) && (a>c))
                printf("A is greater\n");
        else if((b>a) && (b>c))
                printf("B is greater\n");
        else if((c>a) && (c>b))
                printf("C is greater\n");
        else
                printf("All are equal\n");
}
```

Output: A is greater

3. SWITCH STATEMENT

✓ C language provides a multi-way decision statement so that complex else-if statements can be easily replaced by it. C language's multi-way decision statement is called **switch**.

General syntax of switch statement is as follows:

```
switch(choice)
{
                case label1:    block1;
                                break;
                case label2:    block2;
                                break;
                case label3:    block-3;
                                break;
                default:        default-block;
                                break;
}
```

- ✓ Here *switch, case, break and default* are built-in C language words.
- ✓ If the choice matches to label1 then block1 will be executed else if it evaluates to label2 then block2 will be executed and so on.
- ✓ If choice does not matches with any case labels, then default block will be executed.

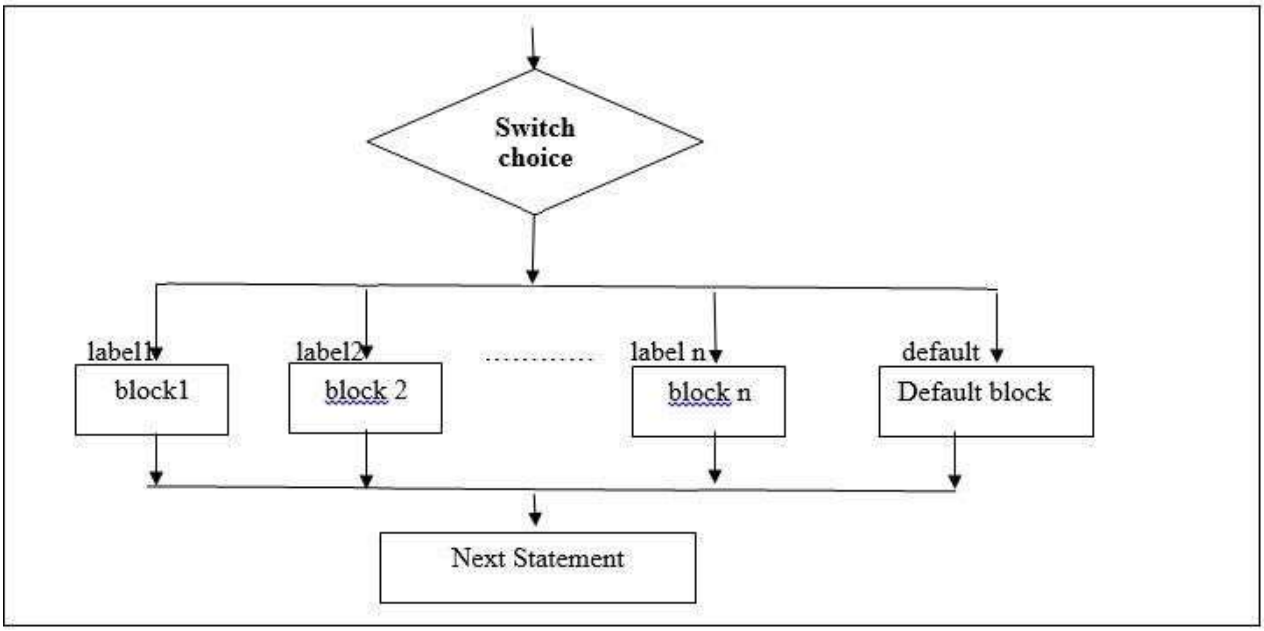


Figure 5: Flow chart of switch case statement

- ✓ The choice is an integer expression or characters.
- ✓ The label1, label2, label3,…. are constants or constant expression evaluate to integer constants.
- ✓ Each of these case labels should be unique within the switch statement. block1, block2, block3, … are statement lists and may contain zero or more statements.
- ✓ There is no need to put braces around these blocks. Note that case labels end with colon(:).
- ✓ Break statement at the end of each block signals end of a particular case and causes an exit from switch statement.
- ✓ The default is an optional case when present, it will execute if the value of the choice does not match with any of the case labels.

Example:

Label → Number	Label → Character

#include<stdio.h>
#include<stdlib.h>
void main( )
{
int ch,a,b,res;
float div;
printf("Enter two numbers:\n");
scanf("%d%d",&a,&b);
printf("1.Addition\n 2.Subtraction\n 3.Multiplication\n 4.Division\n 5.Remainder\n");
printf("Enter your choice:\n");
scanf("%d",&ch);
switch(ch)
{
case 1: res=a+b;
break;
case 2: res=a-b;
break;
case 3: res=a*b;
break;
case 4: div=(float)a/b;
break;
case 5: res=a%b;
break;
default: printf("Wrong choice!!\n");
}
printf("Result=%d\n",res);
}
	#include<stdio.h>
	#include<stdlib.h>
	void main( )
	{
	int a,b,res;
	char ch;
	float div;
	printf("Enter two numbers:\n");
	scanf("%d%d",&a,&b);
	printf("a.Addition\n b.Subtraction\n c.Multiplication\n d.Division\n e.Remainder\n");
	printf("Enter your choice:\n");
	scanf("%c",&ch);
	switch(ch)
	{
	case 'a': res=a+b;
	break;
	case 'b': res=a-b;
	break;
	case 'c': res=a*b;
	break;
	case 'd': div=(float)a/b;
	break;
	case 'e' : res=a%b;
	break;
	default: printf("Wrong choice!!\n");
	}
	printf("Result=%d\n",res);
	}

In this program if ch=1 case '1' gets executed and if ch=2, case '2' gets executed and so on.

4. TERNARY OPERATOR OR CONDITIONAL OPERATOR (?:)

- ✓ C Language has an unusual operator, useful for making two way decisions.
- ✓ This operator is combination of two tokens ? and : and takes three operands.
- ✓ This operator is known as ternary operator or conditional operator.

Syntax:

Expression1 ? Expression2 : Expression3

Where,
Expression1 is Condition
Expression2 is Statement Followed if Condition is True
Expression3 is Statement Followed if Condition is False

Meaning of Syntax:
1. Expression1 is nothing but Boolean Condition i.e. it results into either TRUE or FALSE
2. If result of expression1 is TRUE then expression2 is executed
3. Expression1 is said to be TRUE if its result is NON-ZERO
4. If result of expression1 is FALSE then expression3 is executed
5. Expression1 is said to be FALSE if its result is ZERO

Example: Check whether Number is Odd or Even

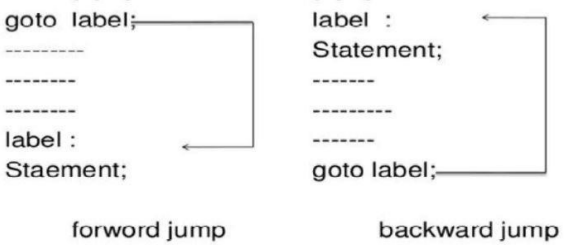
```
#include<stdio.h>
void main()
{
        int num;
        printf("Enter the Number : ");
        scanf("%d",&num);
        flag = ((num%2==0)?1:0);
        if(flag==0)
                printf("\nEven");
        else
                printf("\nOdd");
}
```

5. GOTO STATEMENT

- ✓ goto is an unconditional branching statement. The syntax of goto is as follows:

Syntax	Example

goto *label;*
 	
statement1;
 	
statement2;
 	
label:
	void main( )
	{
	int a=5, b=7;
	goto end;
	a=a+1;
	b=b+1;
	end: printf("a=%d b=%d", a,b);
	}



- ✓ A label is a valid variable name.
- ✓ Label need not be declared and must be followed by colon.
- ✓ Label should be used along with a statement to which control is transferred.
- ✓ Label can be anywhere in the program either before or after the goto label.
- ✓ Here control jumps to *label* skipping statement1 and statement2 without verifying any condition that is the reason we call it unconditional jumping statement.

CHAPTER 3

Decision Making and Looping

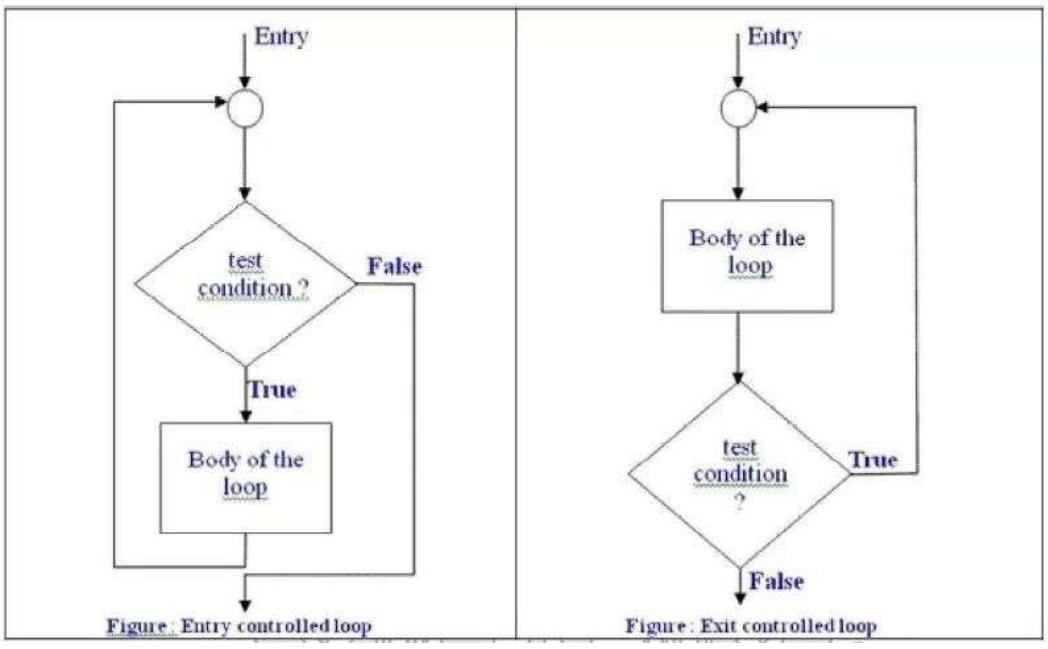
1. INTRODUCTION

Definition of Loop: It is a programming structure used to repeatedly carry out a particular instruction/statement until a condition is true. Each single repetition of the loop is known as an iteration of the loop.

Three important components of any loop are:

1. Initialization (example: ctr=1, i=0 etc)
2. Test Condition (example: ctr<=500, i != 0 etc)
3. Updating loop control values (example: ctr=ctr+1, i =i-1)

Pre-test and Post-test loops



- ✓ Loops can be classified into two types based on the placement of test-condition.
- ✓ If the test-condition is given in the beginning such loops are called pre-test loops (also known as entry-controlled loops).
- ✓ Otherwise if test condition is given at the end of the loop such loops are termed as post-test loops (or exit controlled loops).

Note Figure 1:

1. Here condition is at the beginning of loop. That is why it is called pre-test loop

2. It is also called as entry controlled loop because condition is tested before entering into the loop.

3. while is a keyword which can be used here.

Note Figure 2:

1. Here condition is at the end of the loop. That is why it is called post-test loop.

2. It is also called as exit controlled loop because condition is tested after body of the loop is executed at least once.

3. do and while are keywords which can be used here.

2. LOOPS IN C

C language provides 3 looping structures namely:

1. while loop
2. do….while loop
3. for loop

The loops may be classified into two general types. Namely:

1. Counter-controlled loop
2. Sentinel-controlled loop

✓ When we know in advance exactly how many times the loop will be executed, we use a counter-controlled loop. A counter controlled loop is sometimes called definite repetition loop.

✓ In a sentinel-controlled loop, a special value called a sentinel value is used to change the loop control expression from true to false. A sentinel-controlled loop is often called indefinite repetition loop.

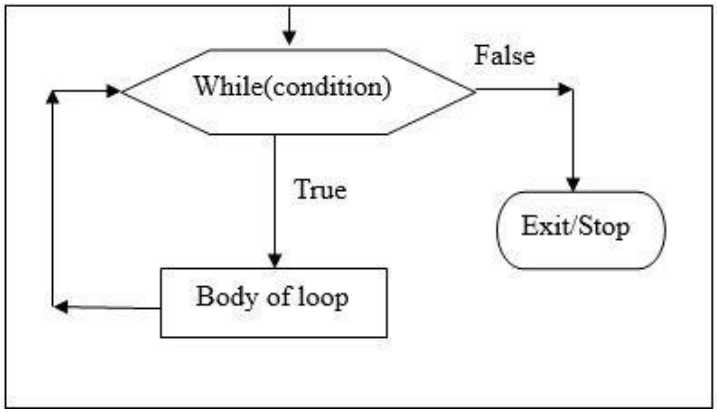
i. while loop:

✓ It is a pre-test loop (also known as entry controlled loop). This loop has following **syntax:**

```
while (condition)
{
    statement-block;
}
```

- ✓ In the syntax given above 'while' is a key word and *condition* is at beginning of the loop.
- ✓ If the test condition is true the body of while loop will be executed.
- ✓ After execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. Minimum number of times executed is 0
- ✓ This process is repeated until condition finally becomes false and control comes out of the body of the loop.

Flowchart:



✓ Here is an example program using while loop for finding sum of 1 to 10.

Example: WAP to find sum of 1 to 5 using while.

```
#include<stdio.h>
void main()
{
        int i=1, sum=0;
        while (i<=5)
        {
                sum=sum+i;
                i=i++;
        }
        printf("%d", sum);
}
```

ii. do…. while loop: It is a post-test loop (also called exit controlled loop) it has two keywords *do* and *while*. The General syntax:

```
do
{
   statement-block;
} while (condition);
```

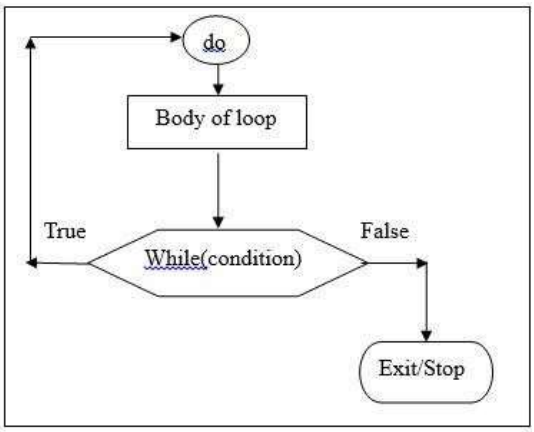
✓ In this loop the body of the loop is executed first and then test condition is evaluated.

✓ If the condition is true, then the body of loop will be executed once again. This process continues as long as condition is true.

✓ When condition becomes false, the loop will be terminated and control comes out of the loop.

Minimum number of times executed is 1

Flowchart:



Example: WAP to find sum of 1 to 5 using do… while

```
#include<stdio.h>
void main()
{
        int i=1, sum=0;
        do
        {
                sum=sum+i;
                i=i++;
        } while (i<=5)
        printf("%d", sum);
}
```

iii. for loop:

- ✓ It is a pre test loop and also known as entry controlled loop.
- ✓ "for" keyword is used here.
- ✓ Here, the head of the for loop contains all the three components that is initialization, condition and Updation.

General syntax:

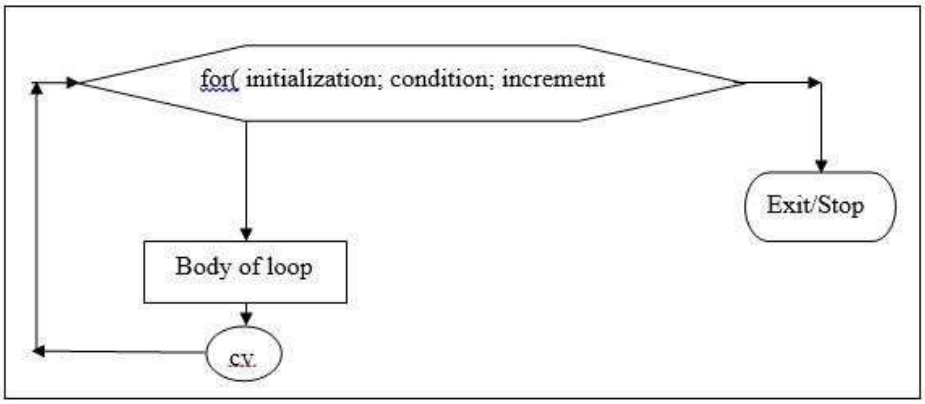
```
for( initialization; test-condition; updation)
{
        Statements;
}
```

Increment/decrement

The execution of for statement is as follows.

1. First the control variable will be initialized. (Example i=1)
2. Next the value of control variable is tested using test condition. (i<=n)
3. If the condition is true the body of the loop is executed else terminated.
4. If loop is executed then control variable will be updated (i++) then, the condition is checked again.

Flowchart:



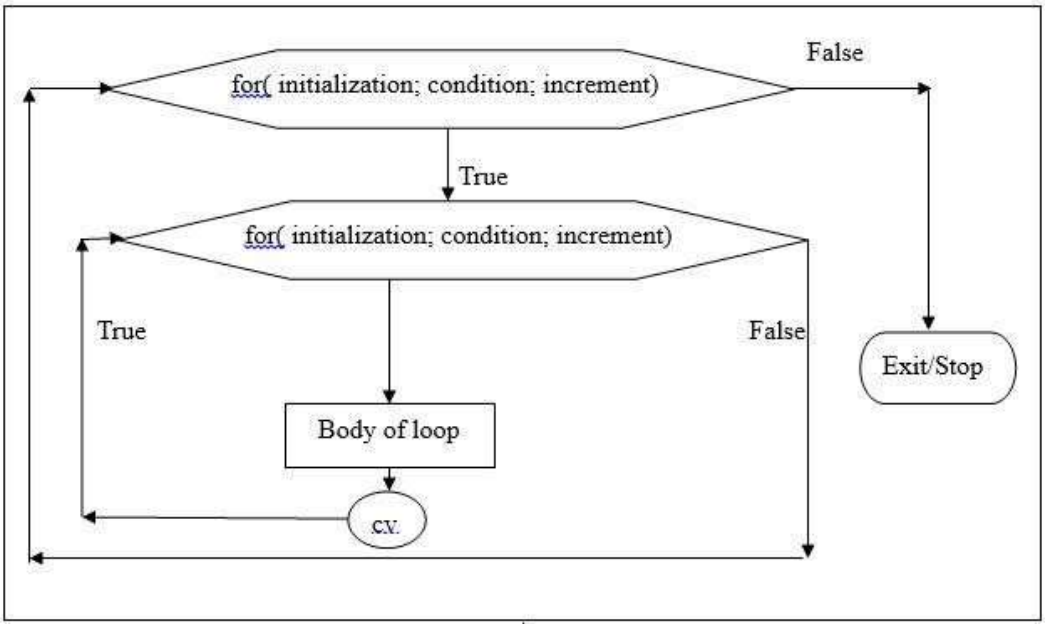
Example: WAP to find the sum of 10 numbers using for loop.

```
#include<stdio.h>
void main( )
{
        int i, sum=0;
        for (i=1;  i<=10; i++)
        {
                sum=sum+i;
        }
        printf("%d", sum);
}
```

Note: In for loops whether both i++ or ++i operations will be treated as pre-increment only.

Nested for loops:

- ✓ A for loop inside a for loop is called Nested for loop.



Example:

```
for(i=0; i<2; i++)
{
        for(j=0; j<2; j++)
        {
                scanf("%d", &a[i][j])
        }
}
```

Note: If updation is not present in loops then, it will execute infinite times.

If initialization is not given then, program prints nothing.

while	do… while
It is a pre test loop.	It is a post test loop.
It is an entry controlled loop.	It is an exit controlled loop.
The condition is at top.	The condition is at bottom.
There is no semi colon at the end of while.	The semi colon is compulsory at the end of while.
It is used when condition is important.	It is used when process is important.
Here, the body of loop gets executed if and only if condition is true.	Here, the body of loop gets executed atleast once even if condition is false.
SYNTAX, FLOWCHART, EXAMPLE (Same as in explanation)	SYNTAX, FLOWCHART, EXAMPLE (Same as in explanation)

3. JUMPS IN LOOPS

✓ **Break and continue:** break and continue are unconditional control construct.

1. Break

✓ It terminates the execution of remaining iteration of loop.
✓ A break can appear in both switch and looping statements.

Syntax

```
while(condition)
{
        Statements;
        if(condition)
                break;
        Statements;
}
```

Flowchart

```
#include<stdio.h>
void main( )
{
        int i;
        for(i=1; i<=5; i++)
        {
                if(i==3)
                        break;
                printf("%d", i)
        }
}
```

Don't want loop to exectute for a particular value

OUTPUT 1 2

2. Continue

- ✓ It terminates only the current iteration of the loop.
- ✓ Continue can appear in looping statements.

Syntax	Flowchart

while(condition)
{
Statements;
if(condition)
continue;
Statements;
}
	#include<stdio.h>
	void main( )
	{
	int i;
	for(i=1; i<=5; i++)
	{
	if(i==3)
	continue;
	printf("%d", i)
	}
	}
	
	OUTPUT 1 2 4 5

Program 1: Computation of Binomial co-efficient

Problem: Binomial coefficients are used in the study of binomial distributions and readability of multicomponent redundant systems. It is given by:

$$B(m,x) = \binom{m}{X} = \frac{m!}{X!(m-X)!}, \quad m >= x$$

A table of binomial coefficients is required to determine the binomial coefficient for any set of m and x.

Problem Analysis: The binomial coefficient can be recursively calculated as follows:

$$B(m,0) = 1$$

$$B(m, x) = B(m, x-1) \left[\frac{m-X+1}{X}\right], \quad X = 1, 2, 3, \ldots, m$$

Further,

$$B(0, 0) = 1$$

That is, the binomial coefficient is one when either x is zero or m is zero. The program below points the table of binomial coefficients for m=10. The program employs one do loop and one while loop.

Program:

```
#include<stdio.h>
#define MAX 10
main( )
{
        int m, x, binom;
        printf(" m x");
        for(m = 0; m <=10; ++m)
                printf(%4d", m);
        printf("\n.....\n");
```

```
        m = 0;
        do
        {
                printf("%2d", m);
                x = 0;
                binom = 1;
                while(x<=m)
                {
                        if(m == 0 || x ==0)
                          printf("%4d", binom);
                        else
                        {
                                binom = binom * (m-x + 1) / x;
                                printf("%4d", binom);
                        }
                        x = x + 1;
                }
                printf("\n");
                m=m+1;
        }
        while (m<= MAX);
        printf("-----------------------------------------------------------\n");
}
```

OUTPUT:

```
mx      0       1       2       3       4       5       6       7       8       9       10
-------------------------------------------------------------------------------------------

0       1
1       1       1
2       1       2       1
3       1       3       3       1
4       1       4       6       4       1
5       1       5       10      10      5       1
6       1       6       15      20      15      6       1
7       1       7       21      35      35      21      7       1
8       1       8       28      56      70      56      28      8       1
9       1       9       36      84      126     126     84      36      9       1
10      1       10      45      120     210     252     210     120     45      10      1
-------------------------------------------------------------------------------------------
```

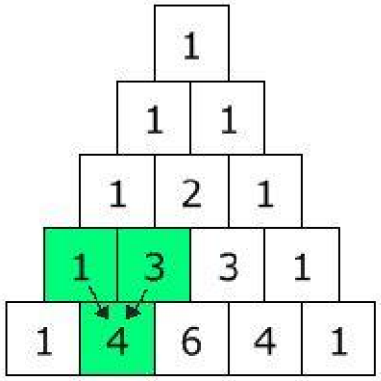
Program 2: Pascal's Triangle

✓ One of the most interesting Number Patterns is Pascal's Triangle (named after *Blaise Pascal*, a famous French Mathematician and Philosopher).

✓ To build the triangle, start with "1" at the top, then continue placing numbers below it in a triangular pattern.

✓ Each number is the numbers directly above it added together.

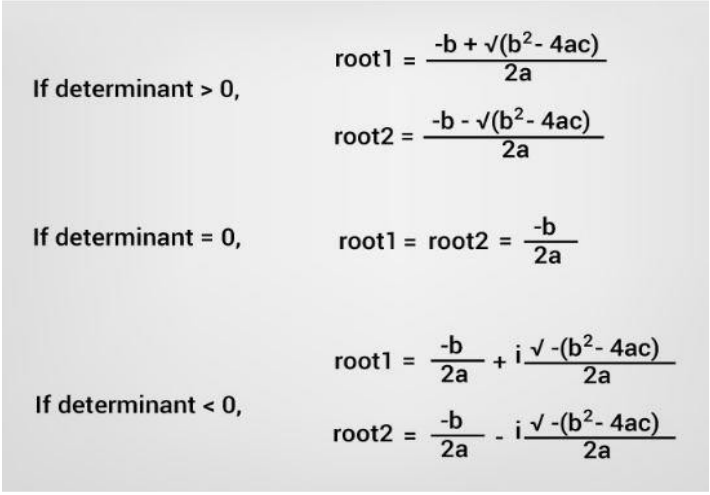
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$



```
#include <stdio.h>

long factorial(int);

int main()
{
  int i, n, c;

  printf("Enter the number of rows you wish to see in pascal triangle\n");
  scanf("%d",&n);

  for (i = 0; i < n; i++)
  {
    for (c = 0; c <= (n - i - 2); c++)
      printf(" ");

    for (c = 0 ; c <= i; c++)
      printf("%ld ",factorial(i)/(factorial(c)*factorial(i-c)));

    printf("\n");
  }

  return 0;
}

long factorial(int n)
{
  int c;
  long result = 1;

  for (c = 1; c <= n; c++)
      result = result*c;

  return result;
}
```

Program 3: Quadratic Equation

✓ The Quadratic Formula uses the "*a*", "*b*", and "*c*" from "$ax^2 + bx + c$", where "*a*", "*b*", and "*c*" are just numbers; they are the "numerical coefficients" of the quadratic equation they've given you to solve.




```
#include<stdio.h>
#include<math.h>
void main()
{
        float a, b, c, root1, root2, rpart, ipart, disc;
        printf("Enter the 3 coefficients:\n”);
        scanf(%f%f%f", &a, &b, &c);
        if((a*b*c) = = 0)
        {
                printf("Roots cannot be Determined:\n");
                exit(0);
        }
        disc=(b*b) - (4*a*c);
        if(disc == 0)
        {
                printf("Roots are equal\n");
                root1= -b / (2*a);
                root2=root1;
                printf ("root1 = %f \n", root1);
                printf ("root2 = %f \n", root2);
        }
        else if(disc>0)
        {
```

```
            printf("Roots are real and distinct\n");
            root1= (-b + sqrt(disc)) / (2*a);
            root2= (-b - sqrt(disc)) / (2*a);
            printf ("root1 = %f \n", root1);
            printf ("root2 = %f \n", root2);
        }
        else
        {
            printf("Roots are complex\n");
             rpart = -b / (2*a);
            ipart = (sqrt (-disc)) / (2*a);
            printf("root1 = %f + i %f \n", rpart, ipart);
            printf("root2 = %f - i %f \n", rpart, ipart);
        }
}
```

Output 1	Output 2	Output 3
Enter the 3 coefficients: 1 2 1 Roots are equal Root1= -1.000000 Root2= -1.000000	Enter the 3 coefficients: 2 3 2 Roots are real and equal Root1=-0.500000 Root2=-2.000000	Enter the 3 coefficients: 2 2 2 Roots are complex Root1=-0.500000+i 0.866025 Root2=-0.500000- i 0.866025

VTU SOLVED QUESTIONS

1. **Evaluate:**

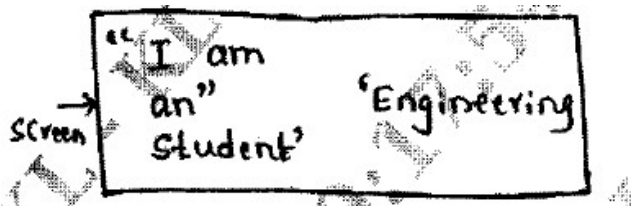
```
i=1
l : if(i >2)
    {
        printf("saturday");
        i = i-1;
        goto l;
     }
     printf("sunday");
```

Explain your result briefly.

Output → Sunday

The i is initialized to 1 and in if condition when checked 1 > 2 returns false. Hence, it doesn't enter inside loop and just prints the Sunday.

2. **Write a C program that prints the following output:**




```
#include<stdio.h>
void main( )
{
        printf("\"I am\n");
        printf("an\" \t 'Engineering\n");
        printf("student'\n");
}
```

3. **State the drawback of ladder if-else. Explain how do you resolve with suitable with suitable example.**

- ✓ If there are more than one if statement and only one else occurs, this situation is called as dangling else problem.
- ✓ This problem is created when no matching else exists for every if.

Ex:

```
if(condition1)
{
}
if(condition2)
{
}
if(condition3)
{
}
else
        printf("Dangling Problem\n");
```

Solutions:

1. The first approach is to follow simple rule i.e., "Always pair on else to most recent unpaired if the current block."

```
if (condition1)
if (condition2)
if(condition3)
else
        printf("Dangling Problem\n");
```

2. The second approach is a compound statement. Here, we make else to belong for the desired if statement using brackets. (In the below example else belongs to first if statement)

```
if (condition1)
{
if (condition2)
if (condition3)
}
else
        printf("Dangling Problem\n");
```

4. **WACP to get the triangle of numbers as a result:**

1
1 2
1 2 3
1 2 3 4

```
#include<stdio.h>
void main( )
{
        int n, i, j;
        printf("Enter the number of rows\n");
        scanf("%d",&n);
        for(i=1 ;i<=n; i++)
        {
                for(j=1; j<=i; j++)
                {
                        printf("%d\t", j);
                }
                printf("\n");
        }
}
```

5. **Write a program in C to display the grade based on the marks as follows:**

Marks	Grades
0 to 39	F
40 to 49	E
50 to 59	D
60 to 69	C
70 to 79	B
80 to 89	A
90 to 100	O

```
#include<stdio.h>
void main( )
{
        int marks;
        printf("Enter the marks\n");
        scanf("%d", &marks);
        if(marks >= 0 && marks <= 39)
                printf("F");
        else if(marks >= 40 && marks <= 49)
                printf("E");
        else if(marks >= 50 && marks <= 59)
                printf("D");
        else if(marks >= 60 && marks <= 69)
                printf("C");
        else if(marks >= 70 && marks <= 79)
                printf("B");
        else if(marks >= 80 && marks <= 89)
                printf("A");
        else
                printf("O");
}
```

6. **WACP to find the sum of natural numbers from 1 to N using while loop.**

```
#include<stdio.h>
void main()
{
        int n, i=1, sum=0;
        printf("Enter the value of n"\n);
        scanf("%d", &n);
        while (i<=n)
        {
                sum=sum+i;
                i=i++;
        }
        printf("%d", sum);
}
```

7. **WAP to find sum of odd numbers from 1 to N using do-while loop.**

```
include<stdio.h>
void main()
{
        int n, i=1, sum=0;
        printf("Enter the value of n"\n);
        scanf("%d", &n);
        do
        {
                sum=sum+i;
                i = i + 2;
        } while (i<=n);
        printf("Sum of odd numbers is %d", sum);
 }
```

8. **WACP to check whether a given number is even or odd using if-else statement.**

```
include<stdio.h>
void main()
{
        int n;
        printf("Enter a number"\n);
        scanf("%d", &n);
        if(n%2==0)
                printf("The given number is Even\n");
        else
                printf("The given number is Odd\n");
 }
```

MODULE-3

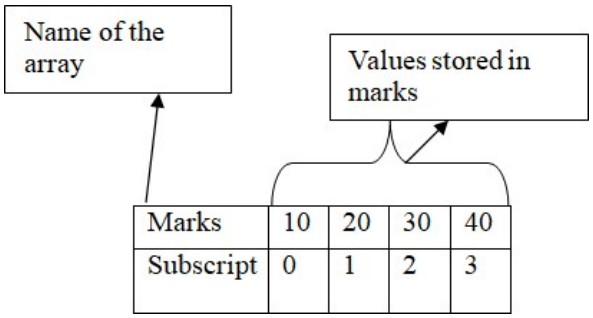
CHAPTER 1: ARRAYS

1. WHY DO WE NEED ARRAYS?

Arrays are used to represent multiple data items of the same type using single name.

2. DEFINITION OF ARRAYS

- The array is a collection of homogeneous elements of same data type.
- The array index starts with 0.
- Arrays are called as subscripted variables because; it is accessed using subscripts/indexes.
- Ex: 1. List of employees in an organization.
  2. List of products and their cost sold by a store.
  3. Test scores of a class of students.



- Here marks[0]=10, marks[1]=20, marks[2]=30 and marks[3]=40. This subscript/index notation is borrowed from Mathematics where we have the practice of writing: $marks_4$, $marks_3$, $marks_2$ and so on.

3. TYPES OF ARRAYS

I. One dimensional array/ Single-Subscripted Variable
II. Two dimensional array/ Double-Subscripted Variable
III. Multidimensional array

I. One-Dimensional Array: A list of items can be given one variable name using only one subscript and such a variable is called as one dimensional array.

Ex: int marks[4];

- Here marks is the name of the array which is capable of storing 4 integer values. The first value will be stored in marks[0], the second value will be stored in marks[1] and so on and the last value will be in marks[3].

Declaration of One-Dimensional Array: Here is general syntax for array declaration along with examples.

Syntax: *data_type array_name[array_size];*

Examples: int marks[*4*];

float temperature[5];

Note: Declaration specifies the data type of array, its name and size.

Initialization of One-Dimensional Array: After array is declared, next is storing values in to an array is called initialization. There are two types of array initialization:

1. Compile-time initialization
2. Run-time initialization

1. Compile time initialization: If we assign values to the array during declaration it is called **compile time** initialization. There are 4 different methods:

a) Initialization with size
b) Initialization without size
c) Partial initialization
d) Initializing all values zero

a) Initialization with size: we can initialize values to all the elements of the array.

Syntax: data_type array_name[array_size]={list of values};

Examples: int marks[4]={ 95,35, 67, 87};

float temperature[3]={29.5, 30.7, 35.6};

b) Initialization without size: We needn't have to specify the size of array provided we are initializing the values in beginning itself.

Syntax: data_type array_name[ ]={list of values};

Examples: int marks[ ]={ 95,35, 67, 87};

float temperature[ ]={29.5, 30.7, 35.6, 45.7, 19.5};

c) Partial initialization: If we not specify the all the elements in the array, the unspecified elements will be initialized to zero.

Example: int marks[5]={10,12,20};

Here, marks[3] and marks[4] will be initialized to zero.

d) Initializing all the elements zero: If we want to store zero to all the elements in the array we can do.

Examples: int marks[4]={0};

2. Run time initialization: Run time initialization is storing values in an array when program is running or executing.

Example:

```
printf("Enter 4 marks");
for(i=0; i<4; i++)
{
        scanf(" %d", &marks[i]);
}
```

II. Two-Dimensional Array: A list of items can be given one variable name using two subscripts and such a variable is called a single subscripted variable or one dimensional array.

Index values

- It consists of both rows and columns. Ex: Matrix.

Declaration of Two-Dimensional Array: Here is general syntax for array declaration along with examples.

Syntax: data_type array_name[row_size][column_size];

Examples: int marks[4][4];

float city_temper[3][3];

Note: Declaration and definition specify the data type of array, its name and size.

Initialization of Two-Dimensional Array: After array is declared, next is storing values in to an array is called initialization. There are two types of array initialization:

1. Compile-time initialization
2. Run-time initialization

1. Compile time initialization: If we assign values to the array during declaration it is called **compile time** initialization. Following are the different methods of compile time initialization.

Syntax: data_type array_name[row_size][column_size]={list of values};

Examples: int marks[3][4]={ 1,2,3, 4, 5, 6, 7,8,9,10,11,12};

float city_temper[2][2]={29.5, 30.7, 35.6, 45.7};

Note: In the above examples we are storing values in marks array and temperature array respectively. The pictorial representation is also given below

- After initialization the arrays appear as follows:

marks

1	2	3	4
5	6	7	8

city_temp

29.5	30.5
35.6	45.7

2. Run time initialization: Run time initialization is storing values in an array when program is running or executing.

Following example illustrates run time storing of values using scanf and for loop:

Example:

```
printf("Enter the marks");
for(i=0; i<3; i++)
for(j=0;j<3;j++)
 {
        scanf(" %d", &marks[i][j]);
}
```

More Examples: Other way of initialization:

```
int a[ ][3]= { 0, 1, 2, 3,4,5,6,7,8};
int  b[ ][4] ={1,2,3,4,5,6,7,8,9,10,11,12};
```

a

0	1	2
3	4	5
6	7	8

b

1	2	3	4
5	6	7	8
9	10	11	12

Example for Invalid initialization

```
int A[3][ ]={1,2,3};
```

Note: Never have column size undeclared in two dimension array.

III. Multi-Dimensional Array: It can have 3, 4 or more dimensions. A three dimensional array is an array of 2D arrays. It has row, columns, depth associated with it.

NOTE:

USING ARRAYS WITH FUNCTIONS:

In large programs that use functions we can pass Arrays as parameters. Two ways of passing arrays to functions are:

1. Pass individual elements of array as parameter
2. Pass complete array as parameter

Pass individual elements of array as parameter	Pass complete array as parameter
Here, each individual element of array is passed to function separately.	Here, the complete array is passed to the function.
Example: #include<stdio.h> int square(int);	**Example**: #include<stdio.h> int sum(int [ ]);

```
int main( )
{
   int num[5], i;
   num[5] ={1, 2, 3, 4, 5};
   for(i=0; i<5; i++)
   {
         square(num[i]);
   }
}
int square(int n)
{
  int sq;
  sq= n * n;
  printf("%d  ", sq);
}
```

```
int main( )
{
   int marks[5], i;
   marks[5] ={10, 20, 30, 40, 50};
   sum(marks);
}
int sum(int n[ ])
{
   int i, sum=0;
   for(i=0; i<5; i++)
   {
      sum = sum+n[i];
   }
   printf("Sum = %d ", sum);
}
```

4. OPERATIONS PERFORMED ON ONE_DIMENSIONAL ARRAY

SORTING: It is the process of arranging elements in the list according to their values in ascending or descending order. A sorted list is called an ordered list.

Ex: Bubble Sort, Selection Sort, Insertion Sort, Shell Sort, Merge Sort, Quick Sort.

Bubble Sort

Bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

```
#include<stdio.h>
void main( )
{
        int n, a[100], i, j, temp;;
        printf("Enter size of array\n");
        scanf("%d",&n);
        printf("Enter elements of array\n");
        for(i=0;i<n;i++)
                scanf("%d", &a[i]);
        for(i=0;i<n;i++)
        {
                for(j=0;j<(n-i)-1;j++)
                {
                        if( a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
```

```
                        }
                }
        }
        printf("The sorted array is\n");
        for(i=0;i<n;i++)
                printf("%d\n",a[i]);
}
```

Selection Sort

Selection sort will start by comparing the first element with other elements and finds minimum element and swaps, then it starts comparing by taking second element with other elements and so on.

```
#include<stdio.h>
void main( )
{
        int n, a[100], i, j, temp, pos;
        printf("Enter size of array\n");
        scanf("%d",&n);
        printf("Enter elements of array\n");
                for(i=0;i<n;i++)
        scanf("%d", &a[i]);
        for(i=0;i<n;i++)
        {
                pos=i;
                for(j=i+1;j<n;j++)
                {
                        if( a[pos]>a[j])
                                pos=j;
                }
                if(pos !=i )
                {
                        temp=a[i];
                        a[i]=a[pos];
                        a[pos]=temp;
                }
        }
        printf("The sorted array is\n");
        for(i=0;i<n;i++)
                printf("%d\n",a[i]);
}
```

SEARCHING: It is the process of finding the location of the specified element in a list.

- The specified element is often called the search key.
- If it is found search is successful, otherwise it is unsuccessful.

Ex: Binary Search, Linear Search, Sequential Search.

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

```
#include<stdio.h>
void main( )
{
    int n, a[100], i, key,loc;
    printf("Enter size of array\n");
    scanf("%d",&n);
    printf("Enter elements of array\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the search element\n");
    scanf("%d",&key);
    loc = -1;
    for(i=0;i<n;i++)
     {
        if(key == a[i])
        {
                loc=i+1;
                break;
        }
    if(loc>=0)
        printf("The element is found at %d \n",loc);
    else
        printf("Search unsuccessful\n");
}
```

Binary Search

Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted. The array is divided into two parts, compared with middle element if it is not successful, then it is compared whether it is lesser or greater than middle element. If it is lesser, search is done towards left part else right part.

```
#include<stdio.h>
void main( )
{
     int n, a[100], i, key, loc, high, low, mid;
    printf("Enter size of array\n");
    scanf("%d",&n);
    printf("Enter elements of array\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the search element\n");
```

```
    scanf("%d",&key);
    loc=-1;
    low=0;
    high=n-1;
    while(low<=high)
    {
        mid=(low+high)/2;
        if(key==a[mid])
        {
            loc = mid+1;
           break;
      }
      else
      {
            if(ele<a[mid])
                high=mid-1;
            else
                low=mid+1;
      }
    }
    if(loc>=0)
        printf("The element is found at %d \n",,loc);
    else
        printf("Search unsuccessful\n");
}
```

CHAPTER 2

STRINGS

1. INTRODUCTION

- ✓ A group of characters together is called String.
- ✓ String is always enclosed within double quotes (“ “)
- ✓ String always ends with delimiter (NULL – ‘\0’)
- ✓ **Ex:** a = “CBIT”

Operations on string:

- ✓ Reading and writing a string
- ✓ Combining string together
- ✓ Copying one string to another
- ✓ Comparing string for equality
- ✓ Extracting a portion of string

2. DECLARATION OF A STRING

A String is declared like an array of character.

Syntax: data_type string_name[size];

Example: char name[20];

String size 20 means it can store upto 19 characters plus the NULL character.

3. INITIALIZATION OF A STRING

Syntax: data_type string_name[size] = value;

Example: The String can be initialized in two ways:

i. char name[30] = "SUVIKA";
ii. char name[30] = {'S', 'U', 'V', 'I', 'K', 'A', '\0'};

NOTE:

i. char a[2] = "CPPS";

The above initialization is invalid because, the size is less.

ii. char a[5];

a = "CPPS";

The above initialization is invalid because, in string the declaration and initialization should be done together.

4. LIMITATION OF STRINGS

One string variable cannot be directly assigned to another variable as shown below:

```
char name1[10] = "Hello";
char name2[10];
name2 = name1; //Invalid
```

But, this initialization can be done using a loop and assigning a individual character as shown below:

```
#include<stdio.h>
void main( )
{
        char name1[10] = "Hello";
        char name2[10];
        int i;
        for(i=0; i<5; i++)
        {
                name2[i] = name1[i];
        }
        name2[i] = '\0';
}
```

5. READING AND WRITING A STRING

i. Reading a String: To read a string we use %s specifier.

Ex:
```
char name[10];
printf("Enter the name\n");
scanf("%s", name);
```

While reading strings we need not specify address operator because, character array itself is an address.

NOTE:

Edit set conversion code %[ ]

It is used to specify the type of character that can be accepted by scanf function.

Ex 1:
```
char name[20];
scanf("%[0123456789]",  name);
```

Example 1 specifies that it accepts only numbers.

Ex 2:
```
char name[20];
scanf("%[^0123456789]", name);
```

Example 2 specifies that it accepts only alphabets, special symbols, space but it does not accept any number.

Ex 3:
```
char name[20];
scanf("%[A-Z]", name);
```

Example 3 specifies that it accepts only upper case alphabets.

ii. Writing/Printing a String: We can use print function with %s format specifier to print a string. It prints the character until NULL character.

Ex 1:
```
char name[10] = "SUVIKA";
printf("The name is %s", name);
```

Output: The name is SUVIKA

Ex 2:
```
char name[10] = "SUVIKA";
printf("%s", name);
printf("%9.3s", name);
printf("%-10.3s", name);
```

Output:

S	U	V	I	K	A

						S	U	V

S	U	V							

6. STRING MANIPULATION FUNCTIONS

- ✓ C library supports a large number of string handling functions that can be used to carry out many of string manipulation and are stored in header file <string.h>
- ✓ There are mainly 6 string handling functions:

Function	Name	Description
strcpy( )	String Copy	Copies one string to another
strlen( )	String Length	Finds the length of a string
strcmp( )	String Compare	Compares two strings
strcat( )	String Concatenation	It concatenates (combines) two strings.
strncpy( )	String 'n' Copy	Copies left most 'n' characters from source to destination.
strncmp( )	String 'n' Compare	Compares left most 'n' characters from source to destination.

I. strcpy( ) – String Copy

It is possible to assign the value to a string variable using strcpy( ). It allows us to copy one string from one location to another.

Syntax: strcpy(destination, source);

This function has two parameters:
i. destination: A string variable whose value is going to be changed.
ii. source: A string variable which is going to be copied to destination.

Ex:
```
char a[10], b[10];
strcpy(a, "CBIT");
strcpy(b, a);
```

After two calls to strcpy( ) a, b contains CBIT.

II. strlen( ) – String Length

The string length function can be used to find the length of the string in bytes.

Syntax: length = strlen(str);

String length function has one parameter str which is a string. It returns an integer value.

Ex:
```
char str[10] = "CBIT";
int length;
length = strlen(str);    //4
```

Note:
```
str = '\0'
length = strlen(str);    //0
```

III. strcmp( ) – String Compare

It is used to compare two strings. It takes the two strings as a parameter and returns an integer value.

Syntax: result = strcmp(first, second);

result = 0 → first = second
result > 0 → first > second
result < 0 → first < second

Example:
```
int res;
res = strcmp("cat", "car");  //res > 0
res = strcmp("pot", "pot");  //res = 0
res = strcmp("big", "small"); //res < 0
```

IV. strcat( ) – String Concatenate

It is used to concatenate or join two strings together. It has two parameters where the combined string will be stored in the (destination) first parameter.

Syntax: strcat(destination, source);

Example:
```
char first[30] = "Computer";
char last[30] = "Programming";
strcat(first, last);
```

Note: strcat( ) stops copying when it finds a NULL character in the second string.

V. strncmp( ) – String 'n' Compare

It compares upto specify number of characters from the two strings and returns integer value.

Syntax: result = strncmp(first, second, numchars);

result = 0 → first = second
result > 0 → first > second
result < 0 → first < second
} → w.r.t number of characters

Example:
```
int res;
res = strncmp("string", "stopper", 4);    //res > 0
res = strncmp("string", "stopper", 2);    //res = 0
res = strncmp("stopper", "string", 4);    //res < 0
```

VI. strncpy( ) – String 'n' Copy

This function allows us to extract a substring from one string and copy it to another location.

Syntax: strncpy(dest, source, numchars);

The strncpy( ) takes three parameters. It copies the number of characters (numchars) from source string to destination string.
Since, numchars doesn't include NULL character we have to specify explicitly.

Ex:
```
char a[10] = "CBIT";
char  b[10];
strncpy(b, a, 2);
b[2] = '\0'
```

7. ARRAY OF STRINGS

It is an array of 1D character array which consists of strings as its individual elements.

Syntax: char name[size1][size2];
Ex: char days[7][10] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};

VTU SOLVED QUESTIONS

1 **WACP to find the transpose of a given matrix.**

```
#include<stdio.h>
void main( )
{
        int m, n, I, j, a[100][100], t[100][100];
        printf("Enter the order of matrix\n");
        scanf("%d%d", &m, &n);
        printf("Enter the elements of matrix\n");
        for(i=0; i<m; i++)
                for(j=0; j<n; j++)
                        scanf("%d", &a[i][j]);
        for(i=0; i<m; i++)
                for(j=0; j<n; j++)
                        t[j][i] = a[i][j];
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                        printf("%d", t[i][j]);
                }
                printf("\n");
        }
}
```

2 **WACP to copy a string (combination of digits and alphabets) to another string (only alphabets).**

```
#include<stdio.h>
#include<ctype.h>
void main()
{
   char s1[100], s2[100];
   int i=0, j=0;
   printf("Enter String1\n");
   scanf("%s",s1);
   while(s1[i] != '\0')
   {
      if(isalpha(s1[i]))
      {
         s2[j] = s1[i];
         j++;
      }
      i++;
   }
   s2[j] = '\0';
```

```
    printf("The copied string is %s\n", s2);
}
```

3 **WACP to find biggest of n numbers using arrays.**

```
#include<stdio.h>
void main()
{
   int n, a[100], i, large=0;
   printf("Enter size of array\n");
   scanf("%d", &n);
   printf("Enter elements of array\n");
   for(i=0; i<n; i++)
   scanf("%d", &a[i]);
   for(i=0; i<n; i++)
   {
      if(a[i] > large)
         large = a[i];
   }
   printf("Large number = %d", large);
}
```

4 **WACP to concatenate two strings without using built in functions.**

```
#include<stdio.h>
void main()
{
   char s1[100], s2[100];
   int i=0, j=0;
   printf("Enter String1\n");
   scanf("%s", s1);
   printf("Enter String2\n");
   scanf("%s", s2);
   while(s1[i] != '\0')
      i++;
   while(s2[j] != '\0')
   {
      s1[i] = s2[j];
      i++;
      j++;
   }
   s1[i] = '\0';
   printf("The concatenated string is %s", s1);
}
```

5 **Differentiate between Linear Search and Binary Search.**

Linear Search	Binary Search
The elements may be in sorted or unsorted order.	The elements must be in sorted order.
It is preferred for small size arrays.	It is preferred for large size arrays.
It is less efficient in case of large size arrays.	It is less efficient in case of small size arrays.
It is based on sequential approach.	It is based on divide and conquer approach.
It finds the position of the searched element by comparing each element.	It finds the position of the searched element by finding the middle element of the array.

MODULE - 4

CHAPTER 1 : USER DEFINED FUNCTIONS

1. INTRODUCTION

Every C program should consist of one or more functions. Among these functions *main( )* function is compulsory. All programs start execution from *main( )* function.

Functions: Functions are independent program modules that are designed to carry out a particular task.

There are two types:

1. Built-in (Library) functions
2. User defined functions

1. ***Built-in functions:*** These are C language functions already available with C compliers and can be used by any programmers.

   Ex: *printf( ),scanf( )*

2. ***User-defined functions:*** These are written by programmers for their own purpose and are not readily available.

Advantages of using Functions:

Functions based modular programming is advantageous in many ways:

1. Managing huge programs and software packages is easier by dividing them into functions/modules—Maintenance is easier
2. Error detection is easier—Debugging is easier
3. Functions once written can be re-used in any other applications – Reusability is enhanced
4. We can protect our data from illegal users—Data protection becomes easier

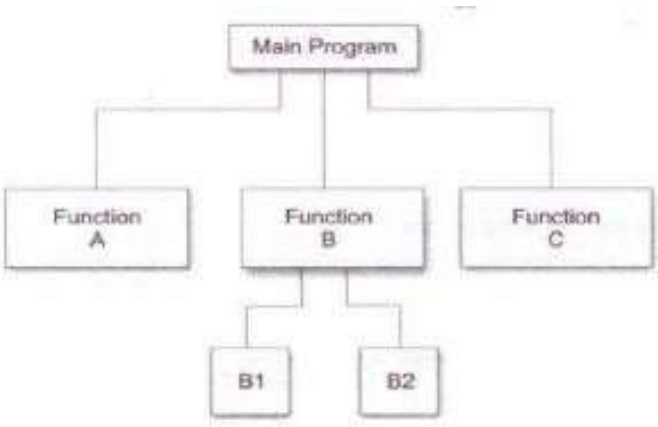
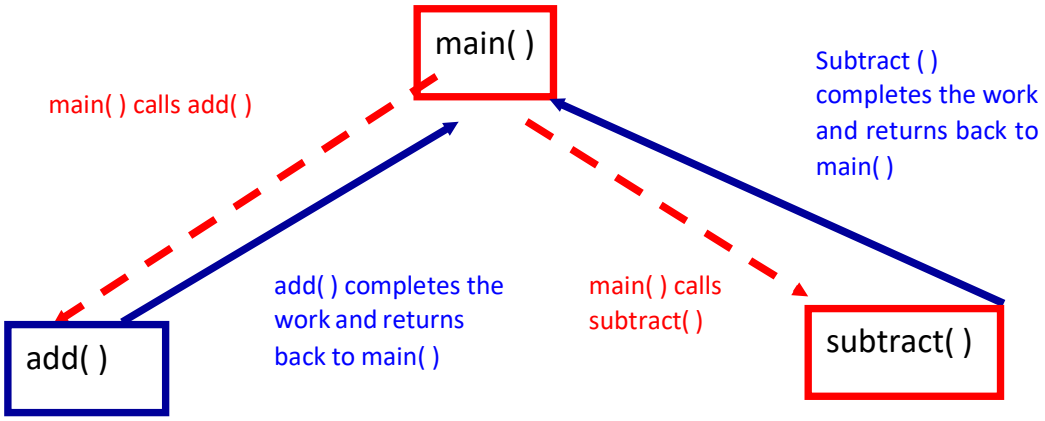


Figure 1: Top-down modular programming using functions

The below block diagram represents how functions are useful: Instead of writing entire program within *main( )* function we can write independent modules as shown:



A function can accept any number of inputs but, returns only one value as output as shown below:

2. MODULAR PROGRAMMING

- ✓ Modular programming is defined as organizing a large program into small, independent program segments called modules that are separately named and individually callable program units.
- ✓ It is basically a "divide-and-conquer" approach to problem solving.
- ✓ The characteristics of modular programming are:
  1. Each module should do only one thing.
  2. Communication between modules is allowed only by calling module.
  3. No communication can take place directly between modules that do not have calling-called relationship.
  4. All modules are designed as single-entry, single-exit systems using control structures.
  5. A module can be called by one and only higher module.

3. ELEMENTS OF USER-DEFINED FUNCTIONS

There are three elements that are related to functions:

i. Function definition
ii. Function call
iii. Function declaration/Function prototype

i. **Function definition:** It is an independent program module that is specially written to implement the requirements of the function.

ii. **Function call:** The function should be invoked at a required place in the program which is called as Function call.

- ✓ The program/function that calls the function is referred as calling program or calling function.
- ✓ The program/function that is called by program/function is referred as called program or called function.

iii. **Function declaration:** The calling program should declare any function that is to be used later in the program which is called as function declaration.

FUNCTION DEFINITION/ DEFINITION OF FUNCTIONS

A function definition, also known as function implementation will include:

i. Function type/ Return type
ii. Function name
iii. List of parameters/ Arguments
iv. Local variable declaration
v. Function statements/ Executable statements
vi. A return statement

All the above six elements are grouped into two parts. Namely:

- Function header (First three elements)
- Function body (Second three elements)

Syntax:

```
function_type  function_name(list of parameters)
{
    local variable declaration;
    executable statement1;
    executable statement2;
    . . . . . .
    . . . . . .
    return statement;
}
```

Note: The first line ***function_type function_name(parameter list)*** is known as the function header and the statements within opening and closing braces is known as the function body.

i. Function Header: It consists of three parts: Function type, function name and list of paraemeters. The semicolon is not present at the end of header.

Function type:

- ✓ The function type specifies the type of value that the function is expected to return to the calling function.
- ✓ If the return type or function type is not specified, then it is assumed as int.
- ✓ If function is not returning anything, then it is void.

Function name: The function name is any valid C identifier and must follow same rules as of other variable.

List of Parameters: The parameter list declares the variables that will receive the data sent by the calling program which serves as input and known as Formal Parameter List.

ii. Function Body: The function body contains the declarations and statements necessary for performing the required task. The body is enclosed in braces, contains three parts:

1. *Local declarations* that specify the variables needed by the function.
2. *Function statements* that perform the task of the function.
3. *A return statement* that returns the value evaluated by the function.

FUNCTION DECLARATION

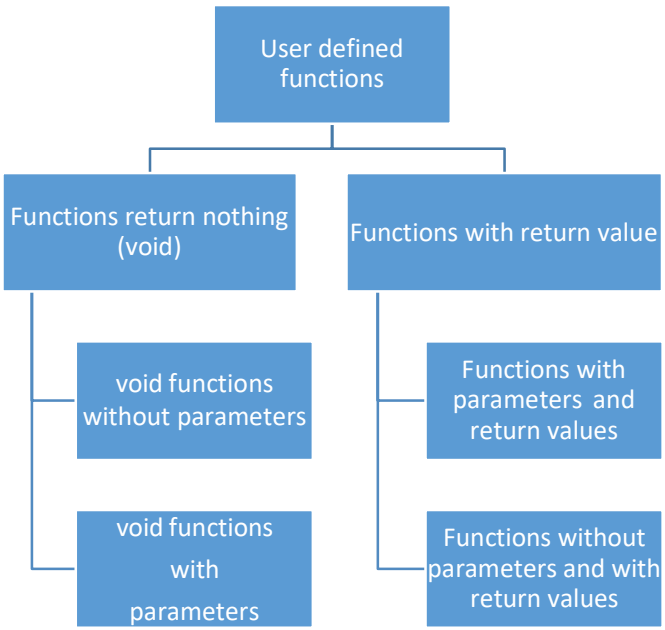
A function declaration consists of four parts:

- Function type/ return type
- Function name
- Parameter List
- Terminating semicolon

Syntax: *function_type function_name(list of parameters);*

Ex: int sum(int, int);
int isprime(int);

4. CATEGORIES OF FUNCTIONS



1. Void Functions without parameters – No arguments and no return values

- ✓ When a function has no arguments, it does not receive any data from the calling function.
- ✓ Similarly, when it does not return a value, the calling function does not receive any data from the called function.

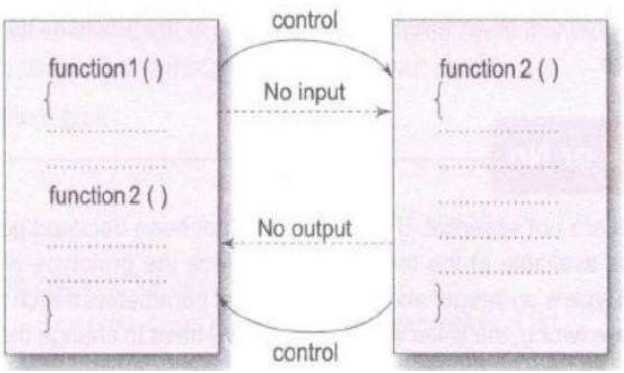


Figure: No data communication between functions

Example:

```
#include<stdio.h>
void add( );
main( )
{
        add( );
}
void add( )
{
        int a=5, b=5, sum=0;
        sum = a+b;
        printf("Result = %d", sum);
        return;
}
```

2. **Void Functions with parameters – Arguments but no return values**

- ✓ The calling function accepts the input, checks its validity and pass it on to the called function.
- ✓ The called function does not return any values back to calling function.
- ✓ The actual and formal arguments should match in number, type and order.

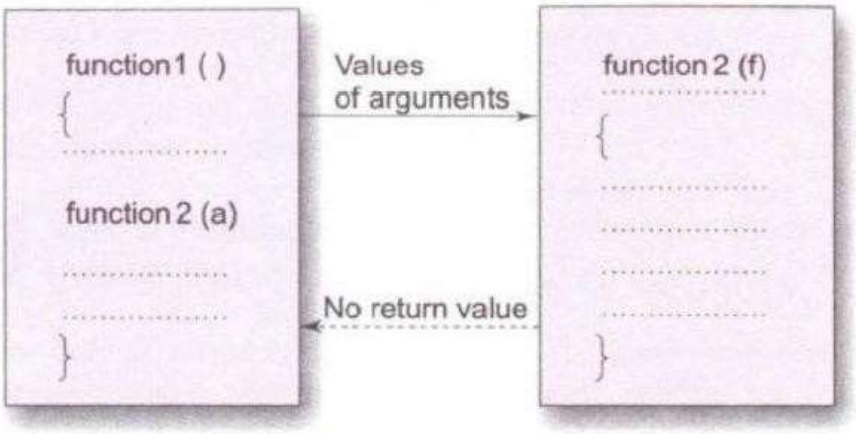


Figure: One-way data communication between functions

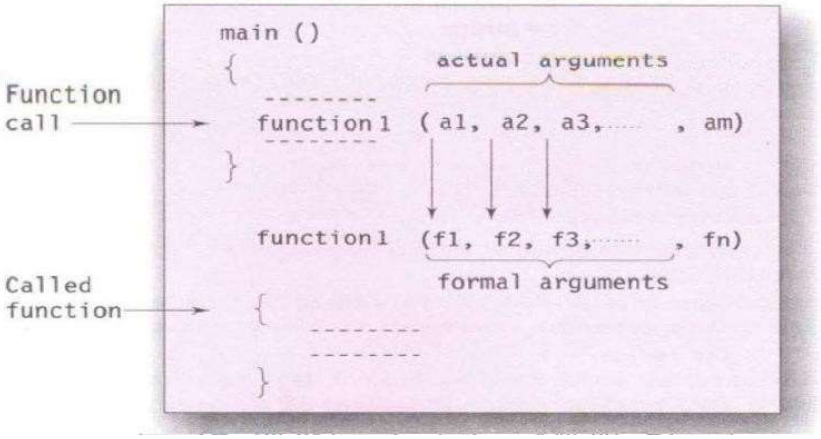


Figure: Arguments matching between the function call and the called function

Example:

```
#include<stdio.h>
void add(int, int);
main( )
{
        int a=5, b=5;
        add(a, b);
}
void add(int a, int b )
{
        int sum=0;
        sum = a+b;
        printf("Result = %d", sum);
        return;
}
```

3. Functions with parameters and return values – Arguments with return values

✓ The calling function sends the data to called function and also accepts the return values from called function.

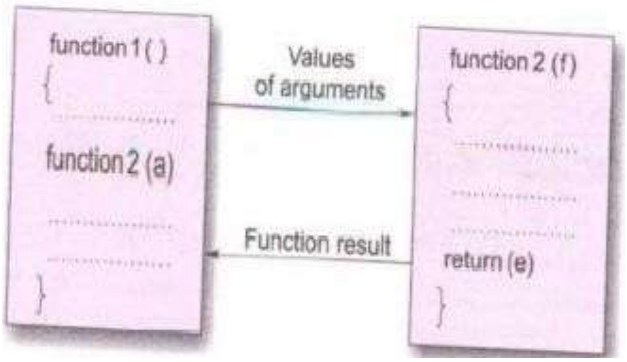


Figure: Two-way data communication between functions

Example:

```
#include<stdio.h>
int add(int, int);
main( )
{
        int a=5, b=5,sum=0 ;
        sum = add(a, b);
        printf("Result = %d", sum);
}
int add(int a, int b)
{
        int x;
        x = a+b;
        return x;
}
```

4. Functions without parameters and with return values – No arguments but return values

- ✓ The calling function does not send any data to called function and but, accepts the return values from called function.

Example:

```
#include<stdio.h>
int add( );
main( )
{
        int sum=0 ;
        sum = add( );
        printf("Result = %d", sum);
}
int add( )
{
        int a=5, b=5, x=0;
        x = a+b;
        return x;
}
```

- ✓ **Actual Parameters:** When a function is called, the values that are passed in the call are called actual parameters.
- ✓ **Formal Parameters:** The values which are received by the function, and are assigned to formal parameters.
- ✓ **Global Variables:** These are declared outside any function, and they can be accessed on any function in the program.
- ✓ **Local Variables:** These are declared inside a function, and can be used only inside that function.

5. INTER-FUNCTION COMMUNICATION

Two functions can communicate with each other by two methods:

1. Pass by value (By Passing parameters as values)
2. Pass by address (By passing parameters as address)

1. Pass by value:

In pass-by-value the calling function sends the copy of parameters (Actual Parameters) to the called function (Formal Parameters). The changes does not affect the actual value.

Example:

```
#include<stdio.h>
int swap(int, int);
int main( )
{
        int a=5, b=10 ;
        swap(a, b);
        printf("%d%d:, a, b);
```

```
}
int swap(int x, int y)
{
        int temp;;
        temp = x;
        x = y;
        y = temp;
}
```

In the above example even though x and y values get swapped, the a and b values remains unchanged. So swapping of two values in this method fails.

2. By Passing parameters as address (Pass by address)

In pass by reference the calling function sends the address of parameters (Actual Parameters) to the called function (Formal Parameters). The changes affects the actual value.

Example:

```
#include<stdio.h>
int swap(int *, int *);
int main( )
{
        int a=5, b=10 ;
        swap(&a, &b);
        printf("%d%d:, a, b);
}
int swap(int *x, int *y)
{
        int temp;;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

In this method the values a and b have been changed or swapped.

6. NESTING OF FUNCTIONS

A function within a function is called as nesting of functions.

Example:

```
#include<stdio.h>
float checkno(float );
float div(float, float);
main( )
{
        float a, b, res;
        printf("Enter two numbers");
        scanf("%f%f",&a,&b);
        div(a,b);
```

```
}
float div(float a, float b)
{
        if(checkno(b))
                printf("Result = %f", a/b);
        else
                printf("Division not possible");
}
float checkno(float b)
{
        if(b!=0)
                return 1;
        else
                return 0;
}
```

7. PASSING ARRAYS TO FUNCTIONS

Two ways of passing arrays to functions are:

1. Pass individual elements of array as parameter
2. Pass complete array as parameter

Pass individual elements of array as parameter	**Pass complete array as parameter**
Here, each individual element of array is passed to function separately.	Here, the complete array is passed to the function.

Example:
 	
#include<stdio.h>
int square(int);
int main( )
{
int num[5], i;
num[5] ={1, 2, 3, 4, 5};
for(i=0; i<5; i++)
{
square(num[i]);
}
}
int square(int n)
{
int sq;
sq= n * n;
printf("%d ", sq);
}
	Example:
	
	#include<stdio.h>
	int sum(int [ ]);
	int main( )
	{
	int marks[5], i;
	marks[5] ={10, 20, 30, 40, 50};
	sum(marks);
	}
	int sum(int n[ ])
	{
	int i, sum=0;
	for(i=0; i<5; i++)
	{
	sum = sum+n[i];
	}
	printf("Sum = %d ", sum);
	}

8. PASSING STRINGS TO FUNCTIONS

The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.

Basic rules are:

1. The string to be passed must be declared as a formal argument of the function when it is defined.
   Ex: void display(char item_name[])
   {
   . . . . . .
   . . . . . .
   }
2. The function prototype must show that the argument is a string.
   Ex: void display(char str[])
3. A call to the function must have a string array name without subscripts as its actual argument.
   Ex: display(names);
   Where names is a properly declared string in the calling function.

CHAPTER 2

RECURSION

Programmers use two approaches to write repetitive algorithms:

- One approach is to use loops
- The other uses recursion

- ✓ A function calling itself repeatedly is called Recursion.
- ✓ All recursive functions have two elements each call either solves one part of the problem or it reduces the size of the problem.
- ✓ The statement that solves the problem is known as base case.
- ✓ The rest of the function is known as general case.

Ex: Recursive definition of factorial is

$$Fact(n) = \begin{cases} 1 & \text{if } n=0 \quad \text{base case} \\ n*fact(n-1) & \text{if } n>0 \quad \text{general case} \end{cases}$$

Limitations of Recursion

- ✓ Recursive solutions may involve extensive overhead because they use function calls.
- ✓ Each time you make a call, you use up some of your memory allocation. If recursion is deep, then you may run out of memory.

PROGRAMS

1. WACP to find a factorial of number using recursion.

```
#include<stdio.h>
int fact(int);
void main( )
{
        int n,res;
        printf("Enter a number");
        scanf("%d",&n);
        res=fact(n);
        printf("Factorial = %d",res);
}
int fact(int n)
{
        if(n==0)
           return 1;
        else
          return (n*fact(n-1));
}
```

2. Program to generate Fibonacci series using recursion

```
#include<stdio.h>
int  Fibonacci(int);
void main()
{
  int n, i = 0, c;
  printf(" Enter the number of terms");
  scanf("%d",&n);
  printf("Fibonacci series\n");
  for ( c = 1 ; c <= n ; c++ )
  {
    printf("%d\n", Fibonacci(i));
    i++;
  }
 }
int Fibonacci(int n)
{
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  else
    return (Fibonacci(n-1) + Fibonacci(n-2));
}
```

VTU SOLVED QUESTIONS

1 **WAP to find GCD and LCM of two numbers using concept of functions.**

```
#include<stdio.h>
int hcf(int, int);
int main( )
{
        int x, y, gcd, lcm;
        printf("Enter two numbers");
        scanf("%d%d", &x, &y);
        gcd = hcf(x, y);
        lcm = (x * y)/ gcd;
        printf("GCD = %d", gcd);
        printf("LCM = %d", lcm);
}
int hcf( int x, int y)
{
        if(x == 0)
                return y;
        while(y!=0)
        {
                if(x > y)
                        x = x-y;
                else
                        y = y-x;
        }
        return x;
}
```

2 **List the storage class specifiers. Explain any one of them.**

OR

Give the scope and lifetime of following:

- **i. External variable**
- **ii. Static variable**
- **iii. Automatic variable**
- **iv. Register variable**

Variable	Definition	Scope	Lifetime
External Variable/ Global Variable	The variables declared outside of all functions.	Global	Runtime of program
Static Variable	The variables declared using static keyword.	Local	Runtime of program
Automatic Variable/ Local Variable	The variables declared inside the function.	Local	Remains within the function in which it is declared.
Register Variable	The variables declared using register keyword.	Local	Remains within the block in which it is declared.

3 **WACP for evaluating the binomial coefficient using a function factorial(n).**

OR

WACP to find the binomial coefficient of a number using recursion.

```
#include<stdio.h>
int factorial(int);
int main( )
{
        int n, r, binom;
        printf("Enter value of n and r");
        scanf("%d%d", &n, &r);
        if(n<0 || r<0 || n<r)
                printf("Invalid Input");
        else
        {
                binom = factorial(n)/ factorial® * factorial(n-r);
                printf("Result = %d", binom);
        }
}
int factorial(int n)
{
        if(n == 0)
                return 1;
        else
                return(n*factorial(n-1));
}
```

4 **Write a C recursive function for multiplying two integers where a function call is passed with two integers m and n.**

```
#include<stdio.h>
int product(int, int);
int main( )
{
        int m, n, res;
        printf("Enter two numbers");
        scanf("%d%d", &m, &n);
        res = product(m, n);
        printf("Result = %d", res);
}
int product(int m, int n)
{
        if(m<n)
                return product(n, m);
        else if(n!=0)
                return (m + product(m, n-1));
        else
                return 0;
}
```

5 **Differentiate:**

i) User defined and Built-in function

ii) ii) Recursion and Iteration

User defined function	Built in function
These are requirement based functions.	These are pre-defined functions.
It is given by users.	It is given by developers.
The names of the functions can be changed.	The names of the functions cannot be changed.
These are part of program called during compile time.	These are part of header file called during run time.
Ex: isprime( ), fibonacci( )	Ex: printf( ), scanf( ), cos( ), sin( )

Iteration	Recursion
Allows set of instructions to be repeatedly executed.	The statement in a body of function calls itself.
Initialization, Condition and Updation are present.	Only termination condition is specified.
Applicable to iterative statements.	Applicable to functions.
The code size is bigger.	The code size is reduced.
No overhead of repeated function calls.	Processes overhead of repeated function calls.

6 **WACP to find the largest element in an array.**

```
#include<stdio.h>
void main( )
{
        int n, a[100], i, large=0;
        printf("Enter the size of array");
        scanf("%d", &n);
        printf("Enter the elements of array");
        for(i=0; i<n; i++)
        scanf("%d", &a[i]);
        for(i=0; i<n; i++)
        {
                if(a[i] > large)
                        large = a[i];
        }
        Printf("Largest number = %d", large);
}
```

7 **WACP using functions to swap two numbers using global variable concept and call by reference concept.**

```
#include<stdio.h>
int a, b;
void swap(int *, int *);
int main( )
{
        printf("Enter two numbers");
        scanf("%d%d", &a, %b);
        printf("Before Swapping a = %d, b= %d", a, b);
        swap(&a, &b);
        printf("After Swapping a = %d, b= %d", a, b);
}
```

```
void swap(int *p, int *q)
{
        int temp;
        temp = *p;
        *p = *q;
        *q = temp;
}
```

MODULE - 5
CHAPTER 1: STRUCTURES

1. INTRODUCTION

Structure: It is a collection of heterogeneous elements of different data type.

Arrays	Structures
Array is a collection of homogeneous elements of same data type.	Structures is a collection of heterogeneous elements of different data type.
Array is a derived data type.	Structure is a programmed defined type.
Arrays must be declared.	Structures must be designed and declared.

2. DEFINING A STUCTURE

The general format of structure is given below:

Synatx

```
struct tagname
{
        datatype member 1;
        datatype member 2;
        -   - - - - - -
        -    - - - - - -
        datatype member n;
};
```

- ✓ The keyword struct defines a structure.
- ✓ tagname/ structure tag indicates name of structure.
- ✓ The structure is enclosed within a pair of flower brackets and terminated with a semicolon.
- ✓ The entire definition is considered as statements.
- ✓ Each member is declared independently for its name and type.

Example 1	Example 2

struct bookbank
{
char author[50];
char title[50];
int year;
float price;
};
	struct student
	{
	char name[50];
	int age;
	float marks;
	};

✓ In the above example 1 bookbank is structure name. ✓ The title, author, price, year are structure members.	✓ In the above example 2 student is structure name. ✓ The name, age, marks are structure members

3. DECLARING A STUCTURE VARIABLE

- ✓ After defining a structure format we can declare variable for that type.
- ✓ It includes the following elements:

i. Keyword struct
ii. A structure or tagname
iii. List of variables separated by comma
iv. Terminating semicolon

Syntax: struct tagname list_of_variables;

Example: struct bookbank book1, book2;

struct student s1, s2;

NOTE:

The complete declaration looks like either of below:

	Example	Syntax

1	struct bookbank
	{
	char author[50];
	char title[50];
	int year;
	float price;
	};
	struct bookbank book1, book2;
		struct tagname
		{
		datatype member 1;
		datatype member 2;
		- - - - - - - -
		- - - - - - - -
		datatype member n;
		};
		struct tagname list_of_variables;

2	struct bookbank { char author[50]; char title[50]; int year; float price; } book1, book2;	struct tagname { datatype member 1; datatype member 2; - - - - - - - - - - - - - - - - datatype member n; }list_of_variables;

4. TYPE DEFINED STRUCTURES

We can use the keyword typedef to define the structures as shown below:

Syntax	Example
typedef struct { datatype member 1; datatype member 2; - - - - - - - - - - - - - - - - datatype member n; }tagname; tagname list_of_variables;	typedef struct { char title[50]; char author[50]; int year; float price; }bookbank; bookbank book1, book2;

5. ACCESSING STRUCTURE MEMBERS

- ✓ The structure members should be linked to the structure variables to make it meaningful.
- ✓ It is done with the help of dot (.) operator.

Ex: i) book1.price → Indicates the price of book1.

ii) book2.author → Indicates the author of book2.

In Programs,

Ex: i) printf("Enter bokk1 price");

scanf("%f", &book1.price);

ii) strcpy(book1.author, "Balaguruswamy");

6. STRUCTURE INITIALIZATION

- ✓ The compile time initialization of a structure variable must have the following elements:
  - ➢ The keyword struct
  - ➢ The structure name or tagname
  - ➢ Name of the variable
  - ➢ Assignment operator (=)
  - ➢ The set of values for the members of the structure variables separated by comma and enclosed in flower brackets
  - ➢ Terminating semicolon

Example

```
struct bookbank
{
        char author[50];
        char title[50];
        int year;
        float price;
};
struct bookbank book1 = {"Balaguruswamy", "CPPS", 2021, 200.0};      //Complete
struct bookbank book2 = {"Kulshreshta", "BE", 2020};                 //Partial
```

The structure can be initialized inside a function or outside a function:

Inside the Function

```
main( )
{
        struct bookbank
        {
                char author[50];
                char title[50];
                int year;
                float price;
        };
        struct bookbank book1 = {"Balaguruswamy", "CPPS", 2021, 200.0};      //Complete
        struct bookbank book2 = {"Kulshreshta", "BE", 2020};                 //Partial
}
```

Outside the Function

```
struct bookbank
{
        char author[50];
        char title[50];
        int year;
        float price;
};
main( )
{
        struct bookbank book1 = {"Balaguruswamy", "CPPS", 2021, 200.0};      //Complete
        struct bookbank book2 = {"Kulshreshta", "BE", 2020};                 //Partial
}
```

Rules for initializing structure

1. We cannot initialize individual member inside the structure template.
2. The order of values must match order of members.
3. It is permitted to have partial initialization.
4. The uninitialized members will be assigned to default values as follows:
   0 → int 0.0 → float '\0' → char

7. COPYING AND COMPARING STRUCTURE VARIABLES

- ✓ The variables of the same structure type can be copied the same way as ordinary variable.
- ✓ If book1 and book2 belong to same structure then,

  book2 = book1 or book1 = book2 → valid.
- ✓ However, C does not permit any logical operation on structure variables like,

  book2 == book1 or book1 = = book2 → Invalid.
- ✓ In case we need to compare them we may do so by comparing members individual.

  book1.price = = book2.price → Valid

8. OPERATIONS ON INDIVIDUAL MEMBERS

The individual members are identified using the member operator (dot .).

A member with the dot operator along with its structure variable can be treated like any other variable name.

Ex:

```
i)    if(book2.year = = 2020)
          book2.price = 500;
ii)   if(book1.price = = 200.0)
          book1.price + = 100;
iii)  sum = book1.price + book2.price;
```

Note:

There are 3 ways to access members. They are:

1. Using dot notation [book1.price]
2. Indirect notation -- Using pointers [*ptr.price]
3. Selection notation – This operator [ptr → price]

9. ARRAY OF STRUCTURES

✓ We can declare an array of structures, each element of the array representing a structure variable.

Example

```
struct bookbank
{
        char author[50];
        char title[50];
        int year;
        float price;
};
struct bookbank book[100];
```

✓ In the above example an array called book is defined, that consist of 100 elements. Each element is defined to be that of type struct bookbank.
✓ It can be initialized as follows:

```
struct bookbank
{
        char author[50];
        char title[50];
        int year;
        float price;
};
struct bookbank book[2] = { {"Balaguruswamy", "CPPS", 2021, 200.0}
                            {"Kulshreshta", "BE", 2020, 500.0} };
```

✓ In the above example it declares the book as an array of 2 elements that is book[0] and book[1].
✓ Each element of book array is a structure variable with 4 members.
✓ An array of structures is stored inside the memory in the same way as a multi-dimensional array as shown in the figure below:

book[0].author	Balaguruswamy
book[0].title	CPPS
book[0].year	2021
book[0].price	200.0
book[1].author	Kulshreshta
book[1].title	BE
book[1].year	2020
book[1].price	500.0

10. ARRAYS WITHIN STRUCTURE

✓ Here, the array is present within structure.

Example:

```
struct marks
{
        int rollno;
        float subject[2];
};
struct marks student[3];
```

✓ In the above example the member subject contains two elements subject[0], subject[1].
✓ The elements can be accessed using appropriate subscript like:
student[0].rollno;
student[0].subject[0];
student[0].subject[1]; → Refers to marks obtained in the 2nd Subject by the 1st Student.

11. STRUCTURES WITHIN STRUCTURE

✓ A structure within a structure is called Nested Structure.

Example

```
struct bookbank
{
        char author[50];
        char title[50];
        struct bookbank1
        {
                int year;
                float price;
        }details;
}book;
```

✓ It can be accessed as:

book.author
book.title
book.details.year
book.details.price

12. STRUCTURES AND FUNCTIONS

✓ There are three methods by which the values of a structure can be transferred from one function to another:
i) The first method is to pass each member of structure as an actual argument of the function.
ii) The second method involves passing of a copy of entire structure to called function.
iii) The third method employs a concept called pointers to pass the structure as an argument.

Function Call Syntax:

```
function_name(structure_variable_name);
```

Function Definition Syntax:

```
data_type function_name(struct_type struct_name)
{
        -   -  -  -  -  -
        -   -  -  -  -  -
        -   -  -  -  -  -
        return(expression);
};
```

CHAPTER 2

POINTERS

1. INTRODUCTION

- ✓ A pointer is a variable which stores the address of another variable.
- ✓ A pointer is a derive data type in 'C'.
- ✓ Pointers can be used to access and manipulate data stored in memory.

2. ADVANTAGES OF POINTERS

- ✓ Pointers are more efficient in handling arrays and data tables.
- ✓ Pointers can be used to return multiple values from a function.
- ✓ Pointers allow 'C' to support dynamic memory management.
- ✓ Pointers provide when efficient tool for manipulating dynamic data structures such as stack, queue etc.
- ✓ Pointers reduce length and complexity of programs.
- ✓ They increase execution speed and this reduces program execution time.

3. UNDERSTANDING POINTERS

Memory Cell	Address
	0
	1
	2
	-
	-
	-
	-
	65535

- ✓ The computer memory is a sequential collection of storage cells as shown in the figure above.
- ✓ The address is associated with a number starting of '0'.
- ✓ The last address depends on memory size.
- ✓ If computer system as has 64KB memory then, its last address is 655635.

4. REPRESENATTION OF A VARIABLE

Ex: int quantity = 179;

- ✓ In the above example quantity is integer variable and puts the value 179 in a specific location during the execution of a program.
- ✓ The system always associate the name "quantity" within the address chosen by system. (Ex: 5000)

Pointer Variables

Variable	Value	Address
quantity	179	5000
P	5000	5048

- ✓ Here, the variable P contains the address of the variable quantity. Hence, we can say that variable 'P' points to the variable quantity. Thus 'P' gets the name Pointer.

NOTE 1:

Pointer Constant: Memory addresses within a computer are referred to as pointer constants. We can't change them but, we can store values in it.

Pointer Values: Pointer values is the value obtained using address operator.

Pointer Variables: The variable that contains pointer value.

NOTE 2:

- ✓ Pointer uses two operators:
  1. The address operator (&)
     It gives the address of an object.
  2. The indirection operator (*)
     It is used to access object the pointer points to.

5. ACCESSING THE ADDRESS OF A VARIABLE

- ✓ The actual location of a variable in the memory is system dependent and therefore the address of a variable is not known to us immediately.
- ✓ Therefore the operator (&) and immediately preceding variable returns the address of the variable associated with it.
- ✓ **Example:**

```
int *quantity;
p = &quantity;
```

6. DECLARING POINTER VARIABLE

Example:

```
int *p                  //Declares a pointer variable p of integer type.
float *sum
```

Syntax: data_type *ptrname;

Where,

data_type → It specifies the type of pointer variable that you want to declare int, float, char and double etc.

*(Asterisk) → Tells the compiler that you are creating a pointer variable.

ptrname → Specifies the name of the pointer variable.

7. INITIALIZATION OF POINTER VARIABLE

✓ The process of assigning address of a variable to a pointer variable is known as Initialization.

Syntax: data_type *ptrname &expression

Where,

data_type → It can be any basic datatype.

ptrname → It is pointer variable

expression → It can be constant value or any variable containing value.

Ex:
```
int a;
int *p = &a;                    // &a is stored in p variable
```

NOTE:

✓ We can initialize pointer variable to NULL or Zero.

```
int *p = NULL;
int *p = 0;
```

8. POINTER FLEXIBILITY

✓ We can make the same pointer to point to different data variables in different statements.

Ex:
```
int x, y, z, *p;
p = &x;
p = &y;
p = &z;
```

✓ We can also use different pointers to point to the same data variable.

Ex:
```
p1 = &x;
P2 = &x;
P3 = &x
```

9. ACCESSING A VARIABLE THROUGH ITS POINTER

✓ After defining a pointer and assigning the address, it is accessed with the help of unary operator asterisk (*) which is called as indirection or dereferencing operator.

Ex:
```
int quantity, n, *p;
quantity = 10;
p = &quantity;
n = *p;
```

10. POINTERS AND STRUCTURES

```
struct bookbank
{
        char author[50];
        char title[50];
        int year;
        float price;
};
struct bookbank book[2], *ptr;
```

- ✓ The above statement declares book as an array of two elements each of the type bookbank and ptr as a pointer to data objects of the type struct bookbank.
- ✓ Therefore, the assignment ptr = &book; would assign address of 0^{th} element of bookbank to ptr that is ptr will point to book[0].
- ✓ Its members can be accessed using " → " (This or arrow operator or member selection operator)
- ✓ **Ex:** ptr → author;
  ptr → title;
  ptr → year;
  ptr → price;

- ✓ To access all the elements of the book below statement is used:

```
for(ptr = book; ptr<book+2; ptr++)
        printf("%s%s%d%f", ptr → author, ptr → title, ptr → year, ptr → price);
```

CHAPTER 3

PRE-PROCESSOR DIRECTIVES

- ✓ Before a 'C' program is compiled the code is processed by program called pre-processor directives. This is called as pre-processing.
- ✓ Commands used in pre-processor are called pre-processor directives and they begin with #.
- ✓ The pre-processor directives in 'C' are:
  i) Macro
  ii) Header file inclusion
  iii) Conditional compilation
  iv) Other directives

i) Macro

This macro function defines constant value and can be of basic data type.

Syntax: #define

Example:
```
#include<stdio.h>
#define PI 3.142
void main( )
{
        float area, r;
```

```
        printf("Enter the radius");
        scanf("%f", &r);
        area = PI * r * r;
        printf("Area = %f", area);
}
```

ii) Header file inclusion

The source code of file "File name" is included in the main program at a specified place.

Syntax: #include<file_name>

Example:

```
#include<stdio.h>
void main( )
{
        int a, b, sum;
        printf("Enter two values");
        scanf("%d%d", &a, &b);
        sum = a+b;
        printf("Sum = %d", sum);
}
```

iii) Conditional Compilation

Set of commands are included or excluded in source program before compilation with respect to the condition.

Ex: #ifdef
#ifndef
#if
#ifelse

#ifdef [if define]

- It tests whether the identifier has been defined or substituted text.
- If the identifier is defined then #ifdef is executed else #else will be executed.

Syntax:

```
#ifdef <identifier>
{
        Statement 1;
        Statement 2;
}
#else
{
        Statement 3;
        Statement 4;
}
#endif
```

Example:

```
#include<stdio.h>
#define LINE 1
void main( )
{
        #ifdef LINE
                printf("LINE ONE");
        #else
                printf("LINE TWO");
        #endif
}
```

OUTPUT: LINE ONE

#ifndef [if not define]

- It tests whether the identifier has been defined or substituted text.
- If the identifier is defined then #else is executed else #ifndef will be executed.

Syntax:

```
#ifndef <identifier>
{
        Statement 1;
        Statement 2;
}
#else
{
        Statement 3;
        Statement 4;
}
#endif
```

Example:

```
#include<stdio.h>
#define LINE 1
void main( )
{
        #ifndef LINE
                printf("LINE ONE");
        #else
                printf("LINE TWO");
        #endif
}
```

OUTPUT: LINE TWO

iv) Other Directives

- **#undef [un-define]:** It is used to undefine a defined macro variable.
- **#pragma:** It is used to call a function before and after main function in a program.

VTU SOLVED QUESTIONS

1 **WACP that finds the addition of two squared numbers, by defining macro for Square(x).**

```
#include<stdio.h>
#define square(x) (x*x)
void main( )
{
        Int n1, n2, sum;
        printf("Enter two numbers");
        scanf("%d%d", &n1, &n2);
        sum = square(n1) + square(n2);
        printf("Sum of two squared numbers = %d", sum);
}
```

2 **WACP that accepts a structure variable as a parameters to a function from a function call.**

```
#include<stdio.h>
#include<string.h>
struct student
{
        int rollno;
        char name[50];
        float percentage;
```

```
};
void func(struct student s1);
int main( )
{
        struct student s1;
        s1.id = 1;
        strcpy(s1.name, “Suvika”);
        s1.percentage = 85.5;
        func(s1);
        return 0;
}
void func(struct student s1)
{
        printf(“ID = %d”, s1.id);
        printf(“Name = %s”, s1.name);
        printf(“Percentage = %f”, s1.percentage);
}
```

3 **WACP to add two numbers using pointers.**

```
#include <stdio.h>
int main()
{
  int n1, n2, *p, *q, sum;
  printf("Enter two numbers");
  scanf("%d%d", &n1, &n2);
  p = &n1;
  q = &n2;
  sum = *n1 + *n2;
  printf("Sum = %d", sum);
  return 0;
}
```

4 **WACP to read details of 10 students and to print the marks of the student if his name is given as input.**

```
#include<stdio.h>
#include<string.h>
struct student
{
        int rollno;
        float marks,
        char name[100], grade[10];
 };
void main()
 {
        struct student s[20];
        int i;
```

```
        char checkname[100];
        for(i=0;i<10;i++)
        {
                printf("Enter the detail of %d students",i+1);
                printf("Enter rollno=");
                scanf("%d",&s[i].rollno);
                printf("Enter marks=");
                scanf("%f",&s[i].marks);
                printf("Enter Name=");
                scanf("%s",s[i].name);
                printf("Enter Grade=");
                scanf("%s",s[i].grade);
        }
        printf("Enter the student name to check the marks");
        scanf("%s", checkname);
        for(i=0;i<10;i++)
        {
            if((strcmp(checkname, s[i].name)) == 0)
                 printf("The marks of the student is %f", s[i].marks);
        }
}
```

5 **Differentiate between Structures and Unions.**

Structures	Unions
struct keyword is used to define a structure.	union keyword is used to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
We can retrieve any member at a time.	We can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

6 **WACP to maintain record of n students using structures with 4 fields (Rollno, marks, name and grade). Print the names of students with marks >= 70.**

```
#include<stdio.h>
 struct student
{
        int rollno;
        float marks,
        char name[100], grade[10];
 };
void main()
 {
        struct student s[20];
        int n,i;
        printf("Enter the number of students");
        scanf("%d",&n);
         for(i=0;i<n;i++)
        {
                printf("Enter the detail of %d students",i+1);
                printf("Enter rollno=");
                scanf("%d",&s[i].rollno);
                printf("Enter marks=");
                scanf("%f",&s[i].marks);
                printf("Enter Name=");
                scanf("%s",s[i].name);
                printf("Enter Grade=");
                scanf("%s",s[i].grade);
        }
        printf("The students who scored above 70 marks are:");
        for(i=0;i<n;i++)
        {
            if(s[i].marks>=70)
                 printf("%s \t %f", s[i].name, s[i].marks);
        }
 }
```

7 **Explain the array of pointers with examples.**

We can declare array of pointers same as of any other data type. The syntax is:

Syntax: data_type *array_name[size];

Ex: int *x[2];

Here, x is an array of 2 integer pointers. It means that this array can hold the address of 2 integer variables.

Ex:
```
int *x[2];
int a=10, b=20;
x[0] = &a;
x[1] = &b;
```