DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

EST 102 PROGRAMMING IN C

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO 1	Analyze a computational problem and develop an algorithm/flowchart to find its solution
CO 2	Develop readable* C programs with branching and looping statements, which uses Arithmetic, Logical, Relational or Bitwise operators.
CO 3	Write readable C programs with arrays, structure or union for storing the data to be processed
CO 4	Divide a given computational problem into a number of modules and develop a readable multi-function C program by using recursion if required, to find the solution to the computational problem
CO 5	Write readable C programs which use pointers for array processing and parameter passing
CO 6	Develop readable C programs with files for reading input and storing output

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓	✓	✓	✓		✓				✓	✓	✓
CO2	✓	✓	✓	✓	✓					✓		✓
CO3	✓	✓	✓	✓	✓					✓		✓
CO4	✓	✓	✓	✓	✓					✓	✓	✓
CO5	✓	✓			✓					✓		✓
CO6	✓	✓			✓					✓		✓

SYLLABUS

EST 102	PROGRAMING IN C	CATEGORY	L	T	P	CREDIT	YEAR OF INTRODUCTION
		ESC	2	1	2	4	2019

Preamble: The syllabus is prepared with the view of preparing the Engineering Graduates capable of writing readable C programs to solve computational problems that they may have to solve in their professional life. The course content is decided to cover the essential programming fundamentals which can be taught within the given slots in the curriculum. This course has got 2 Hours per week for practicing programming in C. A list showing 24 mandatory programming problems are given at the end. The instructor is supposed to give homework/assignments to write the listed programs in the rough record as and when the required theory part is covered in the class. The students are expected to come prepared with the required program written in the rough record for the lab classes.

Module 1

Basics of Computer Hardware and Software

Basics of Computer Architecture: processor, Memory, Input& Output devices

Application Software & System software: Compilers, interpreters, High level and low level languages

Introduction to structured approach to programming, Flow chart Algorithms, Pseudo code (*bubble sort, linear search - algorithms and pseudocode*)

Module 2

Program Basics

Basic structure of C program: Character set, Tokens, Identifiers in C, Variables and Data Types , Constants, Console IO Operations, printf and scanf

Operators and Expressions: Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, size of operator, Assignment operators and Bitwise Operators. Operators Precedence

Control Flow Statements: If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.(Simple programs covering control flow)

Module 3

Arrays and strings

Arrays Declaration and Initialization, 1-Dimensional Array, 2-Dimensional Array

String processing: In built String handling functions (strlen, strcpy, strcat and strcmp, puts, gets) Linear search program, bubble sort program, simple programs covering arrays and strings

Module 4

Working with functions

Introduction to modular programming, writing functions, formal parameters, actual parameters Pass by Value, Recursion, Arrays as Function Parameters structure, union, Storage Classes, Scope and life time of variables, *simple programs using functions*

Module 5

Pointers and Files

Basics of Pointer: declaring pointers, accessing data though pointers, NULL pointer,array access using pointers, pass by reference effect

File Operations: open, close, read, write, append

Sequential access and random access to files: In built file handlingfunctions (*rewind() ,fseek(), ftell(), feof(), fread(), fwrite()*), *simple programs covering pointers and files.*

Text Books

1. Schaum Series, Gottfried B.S.,Tata McGraw Hill,Programming with C
2. E. Balagurusamy, Mcgraw Hill,Programming in ANSI C
3. Asok N Kamthane, Pearson,Programming in C
4. Anita Goel, Pearson, Computer Fundamentals

Reference Books

1. Anita Goel and Ajay Mittal, Pearson, Computer fundamentals and Programming in C
2. Brian W. Kernighan and Dennis M. Ritchie, Pearson, C Programming Language
3. Rajaraman V, PHI, Computer Basics and Programming in C
4. Yashavant P, Kanetkar, BPB Publications, Let us C

Course Contents and Lecture Schedule

Module 1: Basics of Computer Hardware and Software		**(7 hours)**
1.1	**Basics of Computer Architecture:** Processor, Memory, Input& Output devices	**2 hours**
1.2	**Application Software & System software:** Compilers, interpreters, High level and low level languages	**2 hours**
1.3	Introduction to structured approach to programming, Flow chart	**1 hours**
1.4	Algorithms, Pseudo code (*bubble sort, linear search - algorithms and pseudocode*)	**2 hours**
Module 2: Program Basics		**(8 hours)**
2.1	**Basic structure of C program:** Character set, Tokens, Identifiers in C, Variables and Data Types , Constants, Console IO Operations, printf and scanf	**2 hours**
2.2	**Operators and Expressions:** Expressions and Arithmetic Operators, Relational and Logical Operators, Conditional operator, sizeof operator, Assignment operators and Bitwise Operators. Operators Precedence	**2 hours**

2.3	**Control Flow Statements:** If Statement, Switch Statement, Unconditional Branching using goto statement, While Loop, Do While Loop, For Loop, Break and Continue statements.*(Simple programs covering control flow)*	**4 hours**
Module 3: Arrays and strings:		**(6 hours)**
3.1	Arrays Declaration and Initialization, 1-Dimensional Array, 2-Dimensional Array	**2 hours**
3.2	**String processing:** In built String handling functions(*strlen, strcpy, strcat and strcmp, puts, gets*)	**2 hours**
3.3	Linear search program, bubble sort program, *simple programs covering arrays and strings*	**3 hours**
Module 4: Working with functions		**(7 hours)**
4.1	Introduction to modular programming, writing functions, formal parameters, actual parameters	**2 hours**
4.2	Pass by Value, Recursion, Arrays as Function Parameters	**2 hours**
4.3	structure, union, Storage Classes,Scope and life time of variables, *simple programs using functions*	**3 hours**
Module 5: Pointers and Files		**(7 hours)**
5.1	**Basics of Pointer**: declaring pointers, accessing data though pointers, NULL pointer,array access using pointers, pass by reference effect	**3 hours**
5.2	**File Operations**: open, close, read, write, append	1 **hours**
5.3	**Sequential access and random access to files:** In built file handlingfunctions (*rewind() ,fseek(), ftell(), feof(), fread(), fwrite()*), *simple programs covering pointers and files.*	**2 hours**

QUESTION BANK

MODULE I			
Q:NO:	**QUESTIONS**	**CO**	**KL**
1	Differentiate System Software and Application Software.	CO1	K1
2	Draw a flow chart to find the Smallest of 2 numbers and also write its algorithm.	CO1	K2
3	Write a short note on High level Language and Machine Language.	CO1	K4
4	Describe any 4 input devices in detail.	CO1	K5
5	Explain Memory Hierarchy in detail with example.	CO1	K2
6	What are compilers ? Give examples of compiled languages.	CO1	K5
7	Give the difference between algorithm,psuedocode and flow chart.	CO1	K2
8	What do you mean by flow chart ? Give the flow chart symbols in detail.	CO1	K5
9	Draw the flowchart for the program to check whether a number is prime or not.	CO1	K2
MODULE II			
1	Give the difference between between break and continue statements with examples.	CO2	K1
2	Give the difference between while and do while with examples.	CO2	K2
3	What are identifiers ? Give the rules for forming identifiers.	CO2	K4
4	How does X++ differ from ++X ? Explain with suitable examples.	CO2	K5
5	Write the general form of FOR loop. Explain with an example program.	CO2	K2

6	Write a program to find sum and average of 5 numbers.	CO2	K5
7	Write a program to check whether a person is eligible for voting or not.	CO2	K5
8	Explain the concept of escape characters.	CO2	K2
9	Differentiate Increment and Decrement Operators with examples	CO2	K3
	MODULE III		
1	How to declare One-Dimensional Array in C. Give an Example.	CO3	K2
2	Is the statement is true or false. int number[5]={10,20}; Justify your answer.	CO3	K2
3	Explain about strings in C.	CO3	K2
4	How to initialize Two-Dimensional array.	CO3	K4
5	Explain different types of arrays in C. Write its declaration and initialization .	CO3	K5
6	Explain how will you declare and initialize a single dimensional array.	CO3	K2
7	Write the program to implement linear search.	CO3	K2
8	How to declare and initialize a 2-Dimensional array.	CO3	K5
9	Explain the built in string handling functions.	CO3	K2
	MODULE IV		
1	Describe about modular programming.	CO4	K2
2	Describe in detail about pass by value and pass by address	CO4	K2
3	Explain structures in c with suitable program.	CO4	K4
4	Differentiate between structure and union.	CO4	K5
5	Explain the concept of formal and actual parameters.	CO4	K1
6	Explain recursion with example.	CO4	K2

7	Differentiate the storage classes	CO4	K5
8	Explain about scope and life time of variables	CO4	K2
	MODULE V		
1	Define files in C	CO5	K2
2	How pointer variables are initialized	CO5	K4
3	Difference between array and pointer	CO5	K2
4	Difference between an array of pointers and a pointer to an array	CO5	K5
5	Explain an array of pointers	CO5	K2
6	Can main () be called recursively? Explain	CO5	K5
7	Differentiate between *ptr ++ and ++ *ptr	CO5	K2
8	Explain the different file operations in C	CO5	K2

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

S:NO;	TOPIC	PAGE NO:
1	Dynamic memory allocation	174

MODULE NOTES

Module - I

Basics of computer hardware and software.
Basics of Computer architecture : processor, memory, input and output devices. Application software & system software : Compilers, interpreters, high level and low level languages. Introduction to structured approach to programming, flow chart algorithms, pseudocode (bubble sort, linear search - algorithms and pseudocode)

Introduction :

The word "computer" comes from the word "compute", which means to calculate. Hence, a computer is normally considered to be a calculating device, which can perform arithmetic operations at enormous speed.

Computers come in many varieties, including the personal computer, tiny computers built into appliances and automobiles and mainframes machine used by many people simultaneously to run a business.

A complete computer system consists of 4 parts : Hardware, software, people and data.

→ Mechanical devices that make up the computer are called Hardware. In other words hardware is any part of the computer you can touch.

→ Software is a set of electronic instructions consisting of complex codes (also known as programs) that make the computer perform tasks. In other words

software tells the computer what to do.

→ People are the computer operators, also known as users.

→ Data consists of raw facts, which the computer stores and reads in the form of numbers. The computer manipulates the data according to the instructions contained in the software and then forwards it for use by people or another computer. Data can consists of letters, numbers, sounds or images.

→ Within the computer, data is organised into files. A computer file is simply a set of data or program instructions that has been given a name. A file that the user can open and use is often called a document.

Basics of computer : Architecture :

Processor

The procedure that transforms raw data into useful information is called processing. To perform this transformation, the computer uses two components : The processor and memory.

→ The processor is like the brain of the computer in the way that it organises and carries out instructions that come from either the user or the software.

→ In a personal computer, the processor usually consists of one or more microprocessors (sometimes called 'chips'), which are silvers of silicon

for other material etched with many tiny electronic circuits. To process data, the computer passes electricity through the circuits to complete an instruction.

The microprocessor is plugged into the computer's motherboard. The motherboard is a rigid rectangular card containing the circuitry that connects the processor to the other hardware. The motherboard is an example of a circuit board.

In most personal computers, many internal devices such as video cards, sound cards, disk controllers and other devices - are housed on their own smaller circuit boards, which attached to the motherboard.

A personal computer's processor is usually a single chip or a set of multiple chips contained on a circuit board. In some powerful computers, the processor consists of many chips and the circuits boards on which they are mounted.

In either case, the term central processing unit (CPU) refers to a computer's processor.

People often refer to computer systems by the type of CPU they contain. A pentium system, for example uses a pentium class microprocessor as its CPU.

The processor handles all the basic system instructions, such as processing mouse and keyboard input and running applications.

→ most desktop computers contain a CPU dies either intel or AMD both of which use the x86 processor architecture.

Evolution and comparitive study of processors :

A computer processor, generally known as a microprocessor is an electronic circuit which receives the input data from the software and outputs the processed information to various units of a computer system.

(1) The first ever commercially available microprocessor was the Intel 4004 developed by Intel and the Japanese calculator manufacturer Busicom, back in 1971.

Intel 4004 was primarily designed for use in low-performance devices such as calculators, automated teller machines, etc. It consisted of 2300 transistors and was capable of carrying out 92600 operations per second.

(2) The Intel 8008 released in 1972 was the world's first 8-bit microprocessor. This processor came with 3500 transistors and found its use in computers, robots and automated factory machines.

(3) The Intel 8080 was released as a successor to the Intel 8008 in 1974 was one of the most popular microprocessors back in the glory day of computing history.

8-bit microprocessor came with 6000 transistors and was widely used in many microcomputers.

(4) The next major leap in computing performance came with the 16-bit Intel 8086 processor and the slightly improved Intel 8088 which was released in 1979. The Intel 8088 came with - 29,000 transistors on board for faster-operating speeds. This processor was selected for use in IBM PCs.

(5) The Motorola 68000 was one of the high-performance microprocessors in 1979. This 16/32 bit microprocessor was referred to as a main frame on a chip by some owing to its complexity and performance. 68000 transistors used in APPLE Macintosh.

(6) In 1987, Sun microsystems introduced the 1st SPARC (scalable processor architecture) based microprocessors. These were used in high performance computing environments such as mainframes.

(7) Intel released the 1st Intel pentium series processors in 1993. These processors had a whopping 3.1 million transistors and ran at a clock speed of 60 Hz.

(8) The 1st completely home-brewed AMD 5 micro-processor was introduced in 1996.

(9) Forseeing the future of mobile computing, Intel released the Celeron Series microprocessors in 2000.

primarily for use in laptop computers.

(10) IBM was the 1st computing company to introduce multi-core processors, but it was Intel and AMD with their pentium D and Athlon 64 X 2 respectively released in 2005.

(11) In 2008, Intel released the 1st atom series single-core microprocessors to be used in low-cost notebook PCs.

(12) AMD released their 1st mobile-centric processors in 2001 and came with the term Accelerated processing units (APU).

(13) The 1st Ryzen series processors were unveiled by AMD back in 2017.

(14) In 2017, Intel also developed a 12-core desktop CPU for high-performance mobile computing purposes.

(15) The 1st core i9 processor for laptops was released in 2018.

Size of Transistors

Year	Size of Transistors in μm
1971	10
1972	10
1978	3
1993	0.8
2000	0.13
2017	0.014

No. of Transistors & speed.

Year of release	Processors	No: of Transistors	Clock speed
1971	Intel 4004	2300	740 kHz
1972	Intel 8008	6000	800 kHz
1978	Intel 8086	29000	5 MHz
1993	Intel pentium	3.1 million	50 MHz
2000	Intel pentium 4	42 million	1.5 GHz.
2017	AMD Ryzen	4.2 Billion	4.4 GHz.

Memory

Memory is like an electronic scratch pad inside the computer. When you launch a program, it is loaded into and run from memory. Data used by the program is also loaded into memory for fast access.

→ As you enter new data into the computer, it is also stored in memory – but only temporarily.

→ The most common type of memory is called Random Access Memory or RAM.

→ Data is both written to and read from this memory.

→ memory unit consists of cache memory and primary memory [RAM + ROM]

→ Perhaps the most important thing to remember about RAM is that it is volatile, so it needs a constant supply of power. When you turn off a computer, everything in RAM disappears. This is why you frequently have to save your data files to a storage device

→ One of the most important factors affecting the speed and power of a computer is the amount of power it has.

→ Generally, the more RAM a computer has, the more it can do and the faster it can perform certain tasks.

→ The most common measurement unit for describing a computer's memory is the byte - the amount of memory it takes to store a single character, such as a letter or a numeral

→ When people talk about memory, the numbers are often so large that it is useful to use terms such as kilobyte, megabyte, gigabyte and terabyte to describe the values.

Types of RAM

→ There are 2 main types of RAM:

(1) Dynamic RAM (DRAM)

(2) Static RAM (SRAM)

DRAM

DRAM is widely used as a computer's main memory. Each DRAM memory cells is made up of a transistor within an integrated circuit and a data bit is stored in the capacitor.

Since transistors always leak a small amount, the capacitors will slowly discharge, causing information stored in it to drain; hence, DRAM has to be refreshed every few milliseconds to retain data.

→ DRAM is widely used in digital electronics where low-cost and high-capacity memory is required.

Eg: Main memory (RAM) and Graphics cards.

SRAM : Static Random Access Memory

SRAM is made up of CMOS technology and uses 6 transistors. Its construction is comprised of 2 cross-coupled inverters to store data (binary) similar to flip-flops and extra 2 transistors for access control.

→ It is relatively faster than other RAM types such as DRAM.

→ It consumes less power.

→ SRAM can hold the data as long as power is supplied to it.

→ SRAM widely used in cache memory.

UNIT	ABBREVIATION	STORAGE
Bit	B	Binary Digit, Single 1 or 0
Nibble	-	4 bits
Byte / octet	B	8 bits
Kilo byte	KB	1024 bytes
Mega byte	MB	1024 KB
Giga byte	GB	1024 MB
Terabyte	TB	1024 GB
Petabyte	PB	1024 TB
Exabyte	EB	1024 PB
Zetta byte	ZB	1024 EB
Yotta byte	YB	1024 ZB

Secondary memory

→ Secondary memory stores data permanently. Secondary memory also known as external memory.

→ The most common storage medium is the magnetic disk. A disk is a round, flat object that spins around its center. Read/write heads, which are similar to the heads of a tape recorder or VCR, are used to read data from the disk or write data onto the disk.

device that holds a disk is called a disk drive. Some disks are built into the drive and are not meant to be removed; other kinds of drives enable you to remove and replace disks. Most personal computers have at least one non removable hard disk (hard drive) In addition, there is also a diskette drive, which allows you to use removable diskettes. A hard disk can store far more data than a diskette can, so the hard disk serves as the computer's primary filling cabinet. Diskettes are used to load new programs or data onto the hard disk, trade data with other users and make backup copies of the data on the hard disk.

→ Because you can remove diskettes from a computer they are encased in a plastic or vinyl cover to protect them from fingerprints and dust. Because the cover used in early diskettes was thin, the diskette was flimsy or floppy. As a result they came to be called as "floppy disks"

⇒ The CD-Rom drive is the most common type of storage device after the hard and diskette drives.

→ Compact disks (CDs) are a type of optical storage device identical to audio CDs, that can store about 74 minutes of audio or 650 MB of data or about 450 times as much as a diskette.

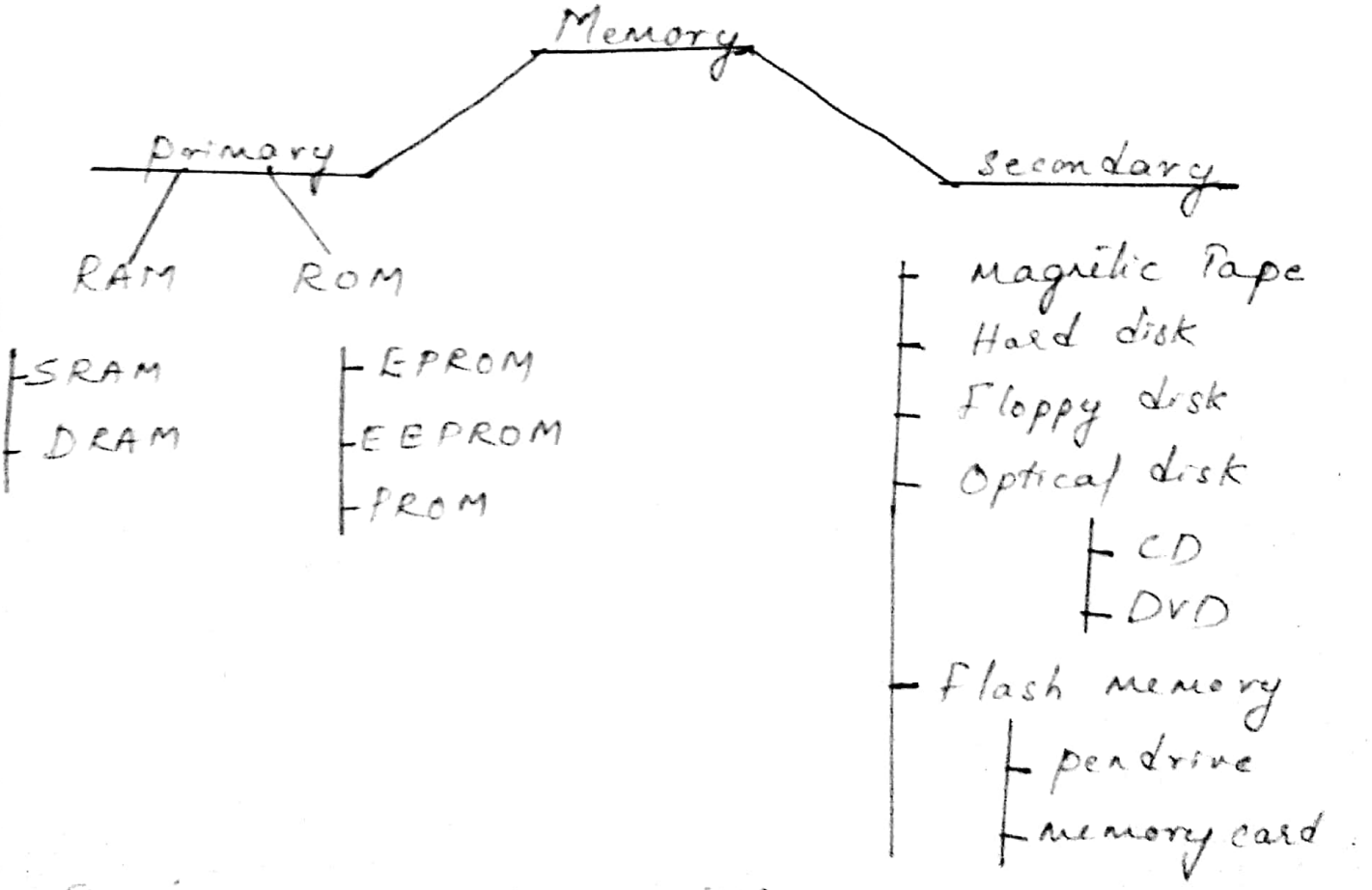
→ The type used in computers is called compact disk Read-only memory (CD-ROM)

→ An increasingly popular data storage technology is the digital versatile disk or digital video disk (DVD) which was revolutionising home entertainment.

→ Using sophisticated compression technologies, a single DVD can store an entire full-length movie.

→ DVD's can hold a minimum of 4.7 GB of data and as much as 17 GB.

memory hierarchy.



Cache Memory

* The data and instructions that are required during the processing of data are brought from the secondary storage devices and stored in the RAM. For processing it is required that the data and instructions are accessed from the RAM and stored in registers. The time taken to move the data between RAM

and cpu registers is large. This affects the speed of processing of computer and results in decreasing the performance of CPU.

* Cache memory is a very high speed memory placed in between RAM and CPU. Cache memory increases the speed of processing.

* Cache memory is a storage buffer that stores the data that is used more often, temporarily and makes them available to CPU at a fast rate. During processing, CPU first checks cache for the required data. If data is not found in cache, then it looks in the RAM for the data.

* Cache memory is built into the processor, and may also be located next to it on a seperate chip between the CPU and RAM. Cache built into the CPU is faster than seperate cache, running at the speed of the microprocessor itself.

* The CPU has a built-in level 1 (L1) cache and level 2 (L2) cache. In addition to the built-in L1 and L2 cache, some CPUs have a seperate cache chip on the motherboard. This cache on the motherboard is called level 3 (L3) cache. Now a days high end processor comes with built-in L3 cache, like in Intel core i7. The L1, L2 and L3 cache store the most recently ran

instructions, the next ones and the possible ones respectively. Typically, CPUs have cache size varying from 256 KB (L1), 6 MB (L2), to 12 MB (L3) cache.

* Cache memory is very expensive, so it is smaller in size. Generally, computers have cache memory of sizes 256 KB to 2 MB.

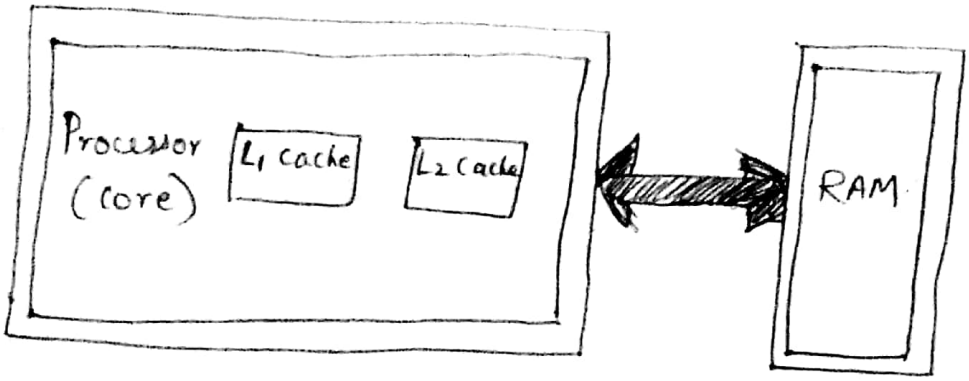


illustration of cache memory.

Primary memory:

* Primary memory is the main memory of computer. It is used to store data and instructions during the processing of data. Primary memory is semiconductor memory.

* Primary memory is of 2 kinds — Random Access Memory (RAM) and Read Only Memory (ROM).

* RAM is volatile. It stores data when the computer is on. The information stored in RAM gets erased when the computer is turned off. RAM provides temporary storage for data and instructions.

ROM is non-volatile memory, but is a read only memory. The storage is ROM is permanent in nature and is used for storing standard processing programs that permanently reside in the computer. ROM comes programmed by the manufacturer.

* RAM stores data and instructions during the execution of instructions. The data and instructions that require processing are brought in to the RAM from the storage devices in which each register is 32 bits wide and its CPU can manipulate 32 bits of data at a time.

Input and Output devices.

A computer interacts with the external environment via the input-output (I/O) devices attached to it. Input device is used for providing data and instructions to the computer. After processing the input data, computer provides output to the user via the output device. The I/O devices that are attached, externally, to the computer machine are also called peripheral devices.

* An I/O unit is a component of computer. The I/O unit is composed of 2 parts - input unit and output unit. The input unit is responsible for providing input to the computer and the output unit is for receiving output from the computer.

INPUT DEVICES:

Input devices allow users and other applications to input data into the computer, for processing. The data input to a computer can be in the form of text, audio, video, etc. The data is entered manually by the user or with minimal user intervention. Input devices are classified as follows:

* Human data-entry devices
  1) keyboard
  2) Pointing devices - mouse, trackball, joystick, digitizing tablet
  3) Pick devices - light pen, touch screen.

* Source data entry devices
  1) Audio input - speech recognition.
  2) Video input - digital camera.
  3) Scanner - hand-held scanner, flat-bed scanner.
  4) Optical scanner - OCR, OMR, MICR, barcode reader.

The input is provided to the computer using an input device and must be translated to a form that the computer can understand. The translation is done by the input interface of the input device.

Output devices:

The output unit gets the processed data from the computer and sends it to the output device to make

& Input

available to the users of computer.

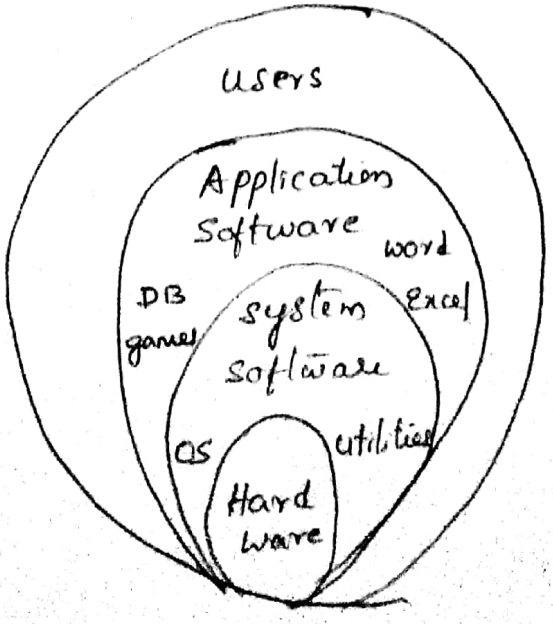
The output data is provided through output devices like display screen, printer, plotter and speaker.

* The processed data sent to the output devices is in a machine understandable form. This processed data is converted to human readable form by the output interface of output device.

In brief, the output unit accepts output data from computer via output device, transforms the output information to human readable form using the output interface of output device and provides the transformed output to user.

Output devices include monitor, visual display terminal, printer, plotter, Audio response system, Video o/p system, Computer output on microfilm (COM).

APPLICATION SOFTWARE & SYSTEM SOFTWARE



System software :

System software provides basic functionality to the computer System software is required for the working of computer itself. The user of computer does not need to be aware about the functioning of system software, while using the computer. For example, when you buy a computer, the System software would also include different device drivers. When you request for using any of the devices, the corresponding device driver software interacts with the hardware device to perform the specified request. If the appropriate particular device drivers for any device, say a particular model of a printer, is installed on the computer, the user does not need to know about the device drivers, while printing on this printer.

The purpose of the system software are

* To provide basic functionality to computer.
* To control computer hardware.
* To act as an interface between user, application software and computer hardware.

Examples :

→ Operating systems : windows, LINUX ----

→ Programming language translators : compilers, debuggers, assemblers.

→ Communication softwares

→ Utility pgms.

Application software:

It is a term which is used for software created for a specific purpose. It is generally a pgm or a collection of programs used by end users. It can be called an application or simply an app.

Examples:

→ Word processing software.
→ Database pgms.
→ Educational software.
→ Business software
→ Entertainment software.

COMPILERS

A compiler is a computer program that transforms code written in a high-level programming language into the machine code. It is the program which translates the human readable code to a language a computer processor understands (binary 1 and 0 bits). The computer processes the machine code to perform the corresponding tasks.

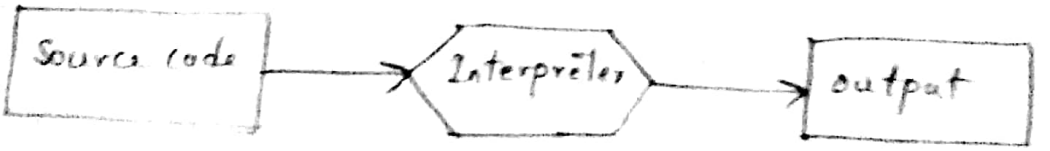
A compiler should comply with the syntax rule of that programming language in which it is written. However, the compiler is only a program and cannot fix errors found in that program. So if you make a mistake, you need to make changes in the syntax of your program. Otherwise, it will not compile.

How compiler works :

Interpreter :

An interpreter is a computer program, which converts each high level program statement into the machine code. This includes source code, pre compiled code, and the scripts. Both compiler and intreprelers do the same job which is converting higher level programming language to machine code. However, a compiler will convert the code into machine code (create an exe) before program run. Interpreters convert code into machine code when the program is run.

How interpreter works



Role of compiler

* Compilers reads the source code, outputs executable code.
* Translates software written in a higher-level language into instructions that computer can understand. It converts the text that a programmer writes into a format the CPU can understand
* The process of compilation is relatively complicated. It spends a lot of time analysing and processing the program

executable result is some form of machine-specific binary code. (11)

Role of Interpreter

* The interpreter converts the source code line by line during RUN Time.
* Interpret completely translates a program written in a high level language into machine level language.
* Interpreter allows evaluation and modification of the pgm while it is executing.
* Relatively less time spent for analysing and processing the program.

Difference between Interpreter and compiler.

Interpreter	Compiler
Translates program one statement at a time	Scans the entire program and translates it as a whole into the machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time but the overall execution time is comparatively faster.
No intermediate object code is generated hence are memory efficient	Generates intermediate code which further requires linking, hence requires more memory.
Continues translating the pgm until the first error is met. Hence debugging is easy.	It generates the error message only after scanning the whole pgm. Hence debugging is hard.
Programming languages like Python, Ruby uses interpreters	Programming languages like C, C++ use compilers.

HIGH - LEVEL LANGUAGE.

A high-level language (HLL) is a programming language such as C, Fortran, or pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages and further from machine languages.

→ **Advantages** of high-level languages

The main advantage of HLL over low-level languages is that they are easier to read, write, and maintain. Ultimately, programs written in a high level language must be translated into machine language by a compiler or interpreter

→ The first high level programming languages were designed in the 1950s. Now there are dozens of different languages like Ada, Algol, BASIC, COBOL, C, C++, JAVA, PROLOG, LISP, PASCAL, FORTRAN, C#, VB.....

Low Level languages:

Low level languages are used to write programs that relate to the specific architecture and hardware of a particular type of computer. They are closer to the native language of a computer (binary), making them harder for programmers to understand.

Examples of low-level languages:

1) Assembly language.
2) Machine code.

Assembly language

Few programmers write programs in low level assembly language, but it is still used for developing code for specialist hardware, such as device drivers. It is easy distinguishable from a high level language as it contains few recognisable human words but plenty of mnemonic code.

Advantages:

* Can make use of special hardware or special machine dependent instructions.
* Translated program requires less memory.
* Write code that can be executed faster.
* Take control over the code.
* Can work directly on memory locations.

Machine code

Programmers rarely write in machine code (binary) as it is difficult to understand. The machine language is normally written as binary 1's and 0's. The circuitary of a computer is wired in a manner that it immediately recognises the machine language instructions and converts them into the electrical signals needed to execute them.

Limitations:

1) Machine dependent.
2) Difficult to program.
3) Error prone.
4) Difficult to modify.

Introduction to structured approach to programming.

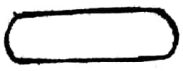
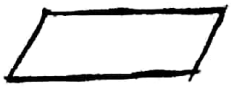
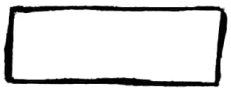
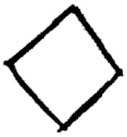
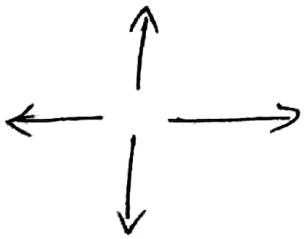
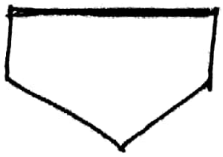
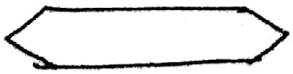
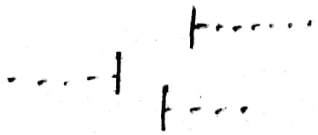
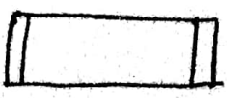
FLOW CHARTS, ALGORITHMS & PSEUDO CODE

FLOW CHARTS:

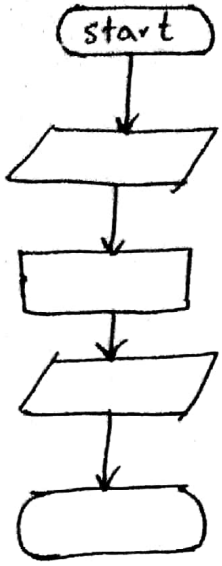
Before you start coding a program it is necessary to plan the step by step solution to the task your program will carry out. Such a plan can be systematically developed using a diagram. This diagram is then called a flow chart. Hence a flow chart is a symbolic representation of a solution to the given task. A flow chart can be developed for practically any job.

Flow chart symbols:

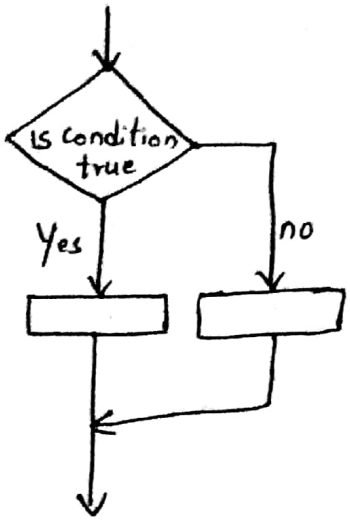
Flow charting has many standard symbols. Flow chart use boxes of different shapes to denote different types of instructions. The actual instruction is written inside the box. These boxes are connected with solid lines which have arrowheads to indicate the direction of flow of the flow chart. The boxes which are used in flowcharts are standardised to have specific meanings. These flowchart symbols have been standardised by American National Standards Institute (ANSI)

Symbol Name	Symbol	Function
Oval		Used to represent start and end of flowchart.
Parallelogram		Used for input & output operation.
Rectangle		Processing: used for arithematic operations & data manipulations.
Diamond		Decision Making: Used to represent the operation in which there are 2/3 alternatives, true and false, etc.
Arrows		Flow line used to indicate the flow of logic by connecting symbols.
Circle		Page Connector.
		Off page connector
		Preprocessor.
		Comments.
		Function/Predefined process. used to represent a group of steps performing one processing task.

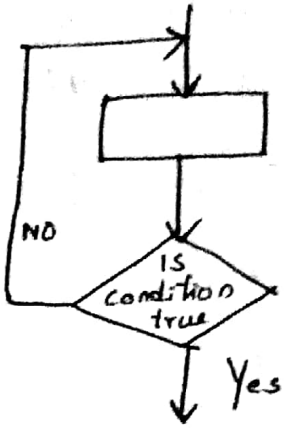
Control Structures in flowchart.



Sequence



Selection



Iteration

In a sequence, the steps are executed in linear order one after the other. In a selection operation, the steps to be executed next is based on a decision taken. If the condition is true (Yes) a different path is followed than if the condition evaluates to false (no). In case of iterative operation, a condition is checked. Based upon the result of this condition check, true or false different paths are followed. Either the next step in the sequence is executed or the control goes back to one of the already executed steps to make a loop.

Algorithms

The word algorithm relates to the name of the mathematician Al-Khowarizmi, which means a procedure or a technique.

An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time.

→ Algorithm has the following characteristics:

- Input : An algorithm may or may not require input.
- Output : Each algorithm is expected to produce atleast one result.
- Definiteness : Each instruction must be clear and unambiguous.
- Finiteness : If the instructions of an algorithm are executed, the algorithm should terminate after the finite number of steps.

DEVELOPMENT OF ALGORITHMS AND FLOWCHART FOR SIMPLE PROGRAMS

- Find the sum of two numbers

Algorithm

step 1 : Start.

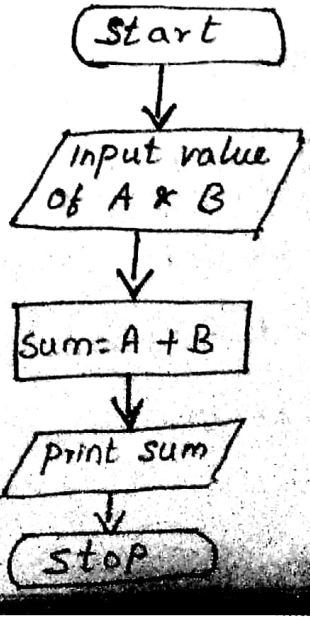
step 2 : Input 2 numbers say A & B.

step 3 : Sum = A + B.

step 4 : Display sum.

step 5 : Stop.

Flowchart



Convert Temperature from Celsius to Fahrenheit

C : Temperature in Celcius

F : Temperature in fahrenheit

ALGORITHM

Step 1 : Start

Step 2 : Input temperature in celcius say C.

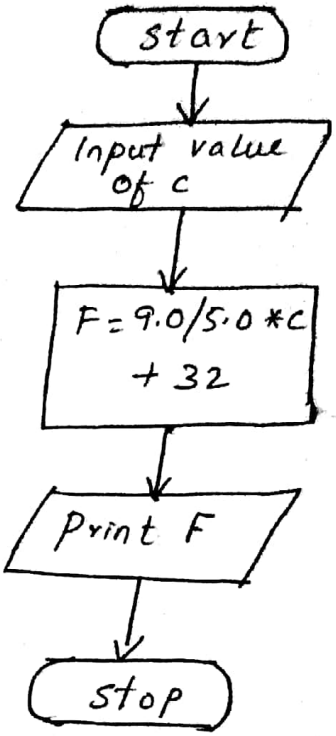
Step 3 : $F = (9.0/5.0 \times C) + 32$.

$C = (5.0/9.0 (F - 32))$

Step 4 : Display temperature in fahrenheit F.

Step 5 : Stop.

FLOWCHART



Find Area and Perimeter of Square

L : Length of square

AREA : Area of Square

PERIMETER : Perimeter of square

Algorithm :

Step 1 : Start

Step 2 : Input length of square, L.

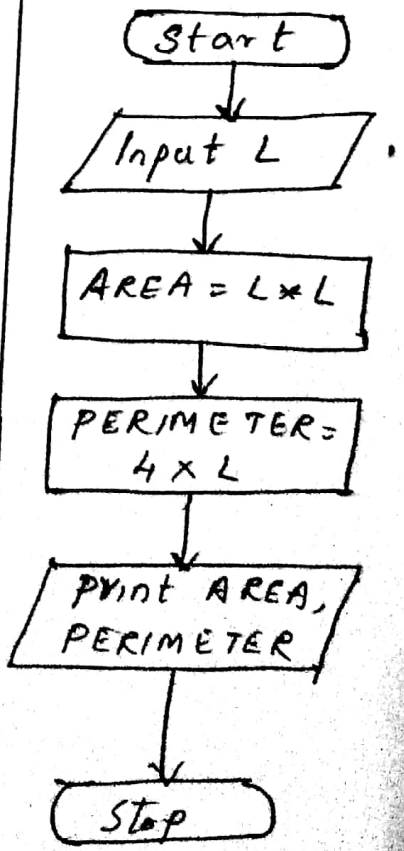
Step 3 : AREA = L × L.

Step 4 : PERIMETER = 4 × L.

Step 5 : Display AREA, PERIMETER.

Step 6 : Stop.

Flowchart



Find the smallest of two numbers

Algorithm

Step 1 : Start

Step 2 : Input 2 numbers say num1, num2.

Step 3 : If num1 > num2 then

print smallest is num1.

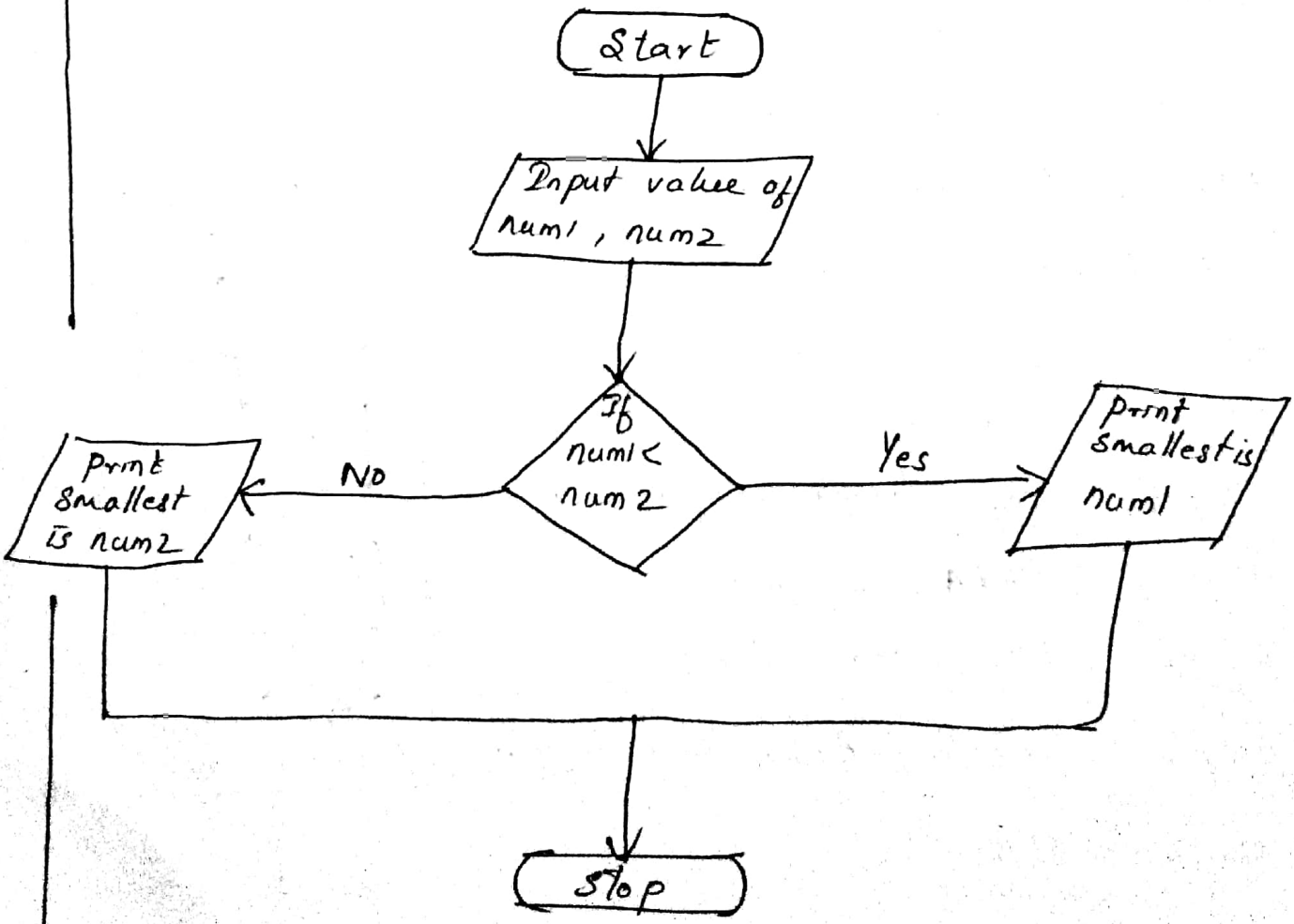
else

print smallest is num2.

endif

Step 4 : Stop.

Flowchart



Pseudocode

Pseudocode is a simple way of writing programming code in english. Pseudocode is not actual programming language. It uses short phrases to write code for pgms before you actually create it in a specific language. Once you know what the program is about and how it will function, then you can use pseudocode to create statements to achieve the required results for your pgm.

Examples of pseudocode.

1) Pseudocode to create a program to add 2 numbers together and then display the result.

```
Start program
Enter two numbers, A, B
Add the numbers together
print sum
End program.
```

2) Pseudocode to compute the area of a rectangle.

```
Get the length, l, and width, w.
Compute the area = l * w
Display the area
```

3) Pseudocode to find the product of any 2 numbers.

```
READ values of A and B
COMPUTE C by multiplying A with B
PRINT the result C
STOP.
```

Module - 2

Program Basics

During 1970s, C had evolved into what is known as "traditional C". The language became more popular after the publication of the book 'The C programming Language' by Brian Kernighan and Dennis Ritchee in 1978. C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts of these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell laboratories, was coded almost entirely in C. To assure that the C language remains standard, in 1983, American National standard Institute (ANSI) appointed a technical committee to define a standard, in 1983, for C. The committee approved a version of C in december 1989 which is now known as ANSI C. It was then approved by the International Standards Organisation (ISO) in 1990. This version of C is also known as C89.

Importance of C.

→ The increasing popularity of C is probably due to its many desirable qualities.

→ It is a robust language whose rich set of built-in functions and operators can used ~~used~~ to write any complex pgms.

→ The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages.

→ Programs written in C are fast and efficient. This is due to its variety of data types and powerful operators.

→ C is highly portable. This means that C programs written for one computer can be run on another computer with little or no modification.

→ Well suited for structured programming, thus requiring the user to think of a problem in terms of functions modules or block.

→ Another important feature of C is its ability to extend itself. A C pgm is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library.

BASIC STRUCTURE OF C PROGRAMS

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

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

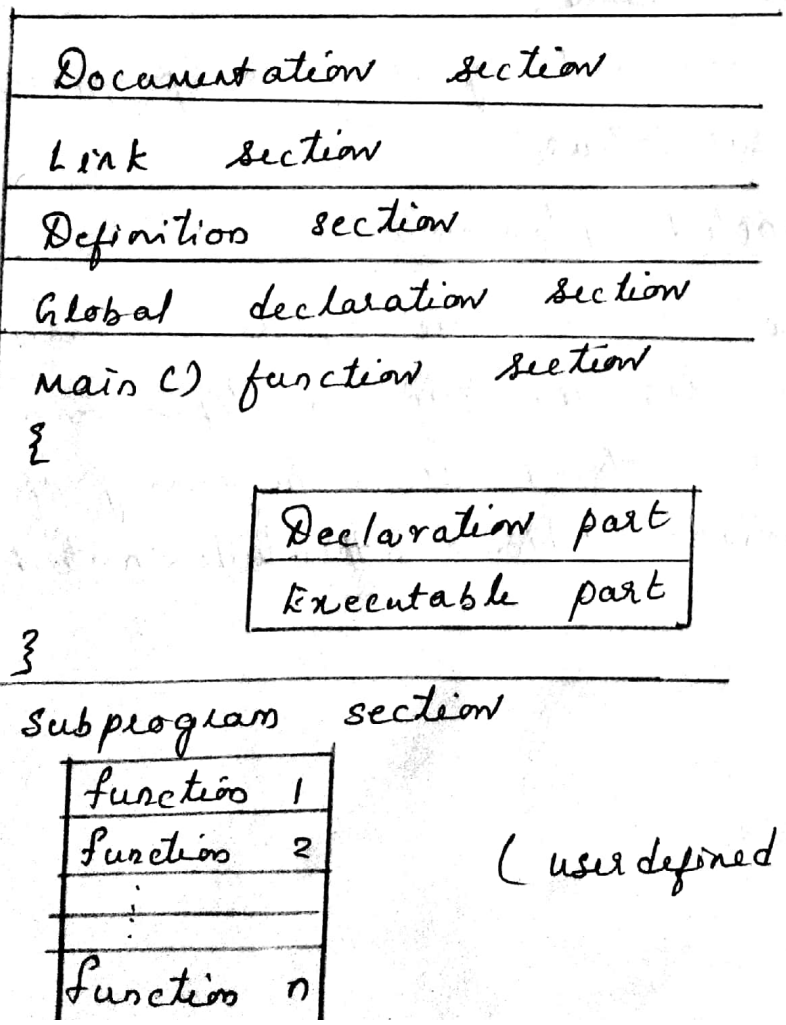
→ The Definition section defines all symbolic constants

→ There are some variables that are used in more than

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

→ Every C program must have one main() function section. This section contains 2 parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program.

→ All statements in the declaration and executable parts ends with a semicolon (;).



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

→ All sections, except the main function section may be absent when they are not required.

THE #include DIRECTIVE

C programs are divided into modules or functions. Some functions are written by users and many others are stored in the C library. Library functions are grouped category wise and stored in different files known as the header files.

→ If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

→ This is achieved by using the preprocessor directive #include as follows:

#include <filename>

→ filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of the program.

→ Eg: #include<stdio.h>, #include<math.h>, #include<conio.h>.

Executing a C program.

Executing a program written in C involved a series of steps:

These are:

1. Creating the program.
2. Compiling the program.
3. Linking the program with functions that are needed from the C library.
4. Executing the program.

UNIX System

→ creating the progeam

The file is created with the help of a text editor, either ed or vi. The command for calling the editor and creating the file is

`ed filename` or `vi filename.c`

Compiling and Linking

→ The compilation command to achieve the compiling of a source program is done by,

`cc filename.c`

The source program instructions are now translated into a form that is suitable for execution by the computer. If everything is alright [checking for correctness], the compilation proceeds silently and

→ The translated program is stored on another file with the name `filename.O`. This is known as object code.

→ Linking process of putting together other program files and functions that are required by the program for example, if the program is using exp() function, then the object code of this function should be brought from the math library of the system and linked to the main program.

→ The compiled and linked program is called the executable object code and is stored automatically in another file named a.out.

→ Executing the program.

The command is `./a.out`

→ MS-DOS SYSTEM

→ Open new file → filename.C

→ Compile → Alt + F9

→ Run → Ctrl + F9.

Constants, variables and Data types

CHARACTER SET

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

1) Letters.
2) Digits.
3) Special characters.
4) White spaces.

Letters

Uppercase A.... Z

Lowercase a.... z

Digits.

All decimal digits 0...9

special characters

, comma
. Period
; Semicolon
: colon
? Question mark
' apostrophe
" quotation mark
! exclaimation mark
| vertical bar
/ slash
\ back slash

~ tilde
_ Underscore
$ dollar sign
% percent
& Ambersand
^ Carcet
- minus sign
+ plus
< opening angle
> Closing angle bracket
(left parenthesis
) Right parenthesis

[left bracket
] Right bracket
{ left brace
} Right brace
number sign

White space
Blank space
Newline.

C Tokens

In a passage of text, individual words & punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens.

C tokens

Keywords	Constants	Strings	Operators
float	-5.5	"ABC"	+ -
while	100	"year"	*

Identifiers

main
amount

special symbols

Keywords and Identifiers

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. All keywords must be written in lower case.

ANSI C KEYWORDS

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

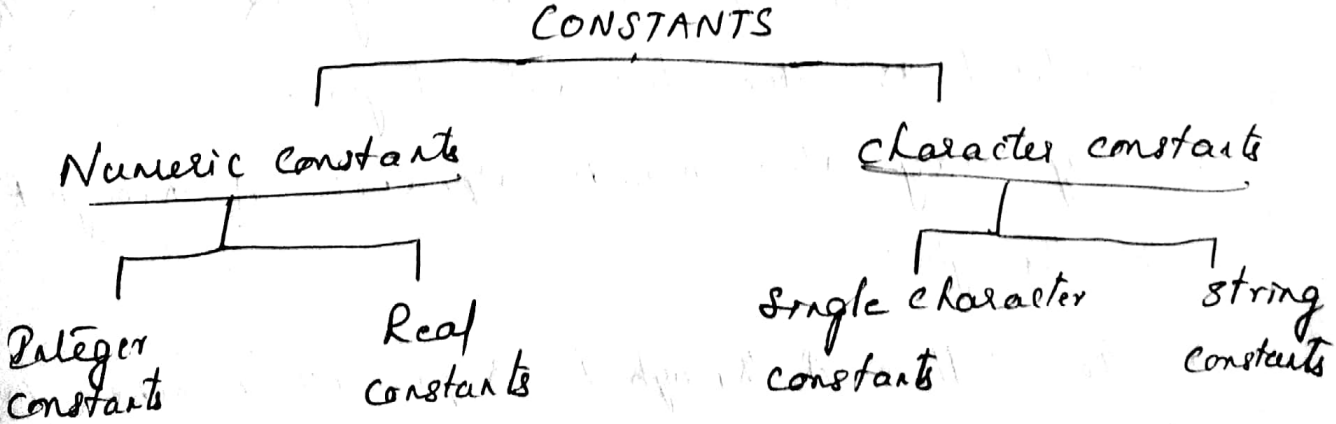
Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identified. It is usually used as a link between 2 words in long identifiers.

Rules for identifiers:

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

☒ CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program.



Basic types of C constants.

① INTEGER CONSTANTS

→ Integer constant refers to a sequence of digits.

→ There are 3 types of integers, namely, decimal integer, octal integer and hexadecimal integer.

→ Decimal integers consists of a set of digits, 0 through 9 preceded by an optional − or + sign.

Examples :

123, -321, 0, +78 ....

→ Embedded spaces, commas and non-digit characters are not permitted b/w digits.

15 750, 20,200, $1000 are illegal numbers.

→ An octal integer consists of any combination of digits from the set 0 through 7, with a leading 0.

→ Examples

037, 0, 0435, 0551.

→ A sequence of digit preceded by 0x or 0x is considered as Hexadecimal integers.

→ They may also include alphabet A through F or a through f.

→ The letters A through F represents numbers 10 through 15.

→ Examples

0x2, 0x9F, 0xbcd

2) REAL CONSTANTS

→ Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like 16.548 Such numbers are called real (or floating point) constants.

→ Examples:

0.083 , −0.75, 435.36

→ These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part.

→ It is possible to omit digits before the decimal point, or digit after the decimal point. That is,

215. , .95 , −.71 , +.5 are valid.

(3) SINGLE CHARACTER CONSTANTS

→ A single character constant contains a single character enclosed within a pair of single quote marks.

→ Examples:

'5', 'x', ';', ' ' → Blank space.

→ The character constant '5' is not the same as the number 5.

→ Character constants have integer values known as ASCII values.

→ Examples: The statement

printf("%d", 'a'); would print the number 97, the ASCII value of 'a'.

printf ("%c", 197); would print the letter 'a'.

4) STRING CONSTANTS

→ A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank spaces

→ Examples:

"Hello!" "1987" "HAI" "5+3"

→ A character constant 'x' is not equivalent to single character string "x".

Backslash character constants

C supports some special backslash character constants that are used in output functions.

Constant	Meaning
'\b'	backspace
'\n'	Newline
'\t'	Horizontal tab
'\v'	Vertical Tab
'\?'	Question mark
'\\'	backslash.

VARIABLES

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

→ A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program.

→ Examples:

Average, height, total, counter_1

→ Variable names may consists of letters, digits, and the underscore (_) character, subject to the following conditions:

i) They must begin with a letter.
ii) ANSI standard recognises a length of 31 characters.
iii) Uppercase & Lowercase are significant. That is, the variable Total is not same as total or TOTAL.
iv) It should not be a keyword.
v) White space is not allowed.

Examples:	Valid variable Names		Invalid variable names		
	John	value	T_raise	123	25th	gP one
	Delhi	x1	Pb_value	%	char	price$
	mark	sum1	distance	(area)		

⊠ Data types

C language is rich in its data types, storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

→ ANSI C support 3 classes of data types:

(1) primary (fundamental) data types

(2) Derived data types

(3) User defined data types.

→ All C compilers support 5 fundamental data types namely integer (int), character (char), floating point (float), double-precision floating point (double) and void.

PRIMARY DATA TYPES

Integral type		
integer		Character
Signed	unsigned type	char
int	unsigned int	signed char
short int	unsigned short int	unsigned char
long int	unsigned long int	

Floating point type		
float	double	long double

void

a) Integer types

→ Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary the size of an integer that can be stored depends on the computer.

→ In order to provide some control over the range of numbers and storage space, C has 3 classes of integer storage, namely short int, int and long int, in both signed and unsigned forms.

→ ANSI C defines these types so that they can be organised from the smallest to the largest.

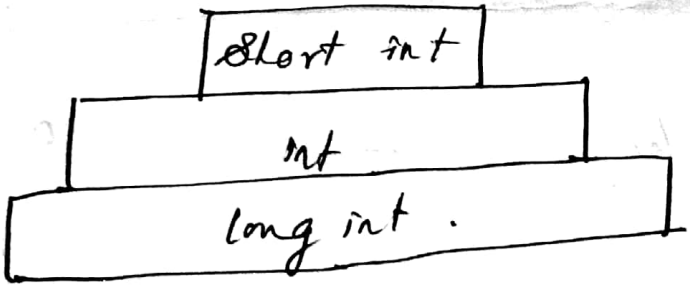


Fig: Integer types.

→ Short int represents fairly small integer values and requires half the amount of storage as a regular int number uses.

→ Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive.

→ Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

→ Size and Range of Data types on a 16-bit machine

Type	Size (bits)	Range
Char or signed char	8	-128 to 127
Unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
Unsigned int	16	0 to 65535
Short int or signed Short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2,147,483,648 to 2,147,483,647
Unsigned long int	32	0 to 4,294,967,29
float	32	3.4E-38 to 3.4E
double	64	1.7E-308 to 1.7E+
long	80	3.4E-4932 to 1.1E+4932

b) Floating point Types

→ Floating point (or real) numbers are stored in 32 bits, with 6 digits of precision.

→ Floating point numbers are defined in C by the keyword float.

→ When the accuracy provided by a float number is not sufficient, the type double can be used to define the number.

→ A double data type number uses 64 bits giving a precision of 14 digits.

→ These are known as double precision number.

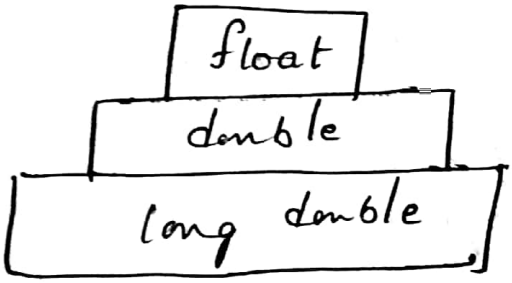


Fig: Floating-point types

→ To extend the precision further, we may use long double which uses 80 bits.

(c) Void types

→ The void type has no values.

→ This is usually used to specify the type of functions.

→ The type of a function is said to be void when it does not return any value to the calling function.

→ It can also play the role of a generic type, meaning that it can represent any of the other standard types.

(d) Character types

→ A single character can be defined as a character (char) type data.

→ Characters are usually stored in 8 bits (one byte) of internal storage.

→ The qualifier signed or unsigned may be explicitly applied to char.

→ Unsigned chars have values between 0 and 255. Signed chars have values from -128 to 127.

Declaration of Variables

Declaration does 2 things:

1) It tells the compiler what the variable name is.
2) It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows

data-type $V_1, V_2, \ldots V_n$;

$V_1, V_2, \ldots V_n$ are the names of variables, variables are seperated by commas. A declaration statement must end with a semicolon. For example, valid declarations are

```
int count;
int number, total;
double ratio;
```

int and double are the keywords to represent integer and real type data values respectively.

Data Types and their Keywords

	Data Types	Keywords equivalent
1	Character	char
2	Unsigned character	unsigned char
3	Signed character	signed char
4	Signed integer	signed int (or int)
5	Signed short integer	signed short int (or short int or short)
6	Signed long integer	signed long int (long int or long)
7	Unsigned integer	unsigned int (or unsigned)
8	Unsigned short integer	unsigned short int (or unsigned short)
9	Unsigned long integer	unsigned long int (or unsigned long)
10	Floating point	float
11	Double-precision floating point	double
12	Extended double-precision floating point	long double.

Assigning values to variables

Assignment statement

values can be assigned to variables using the assignment operator = as follows:

variable_name = constant ;

Eg: balance = 75.84 ;
final_value = 100 ;

To assign a value to a variable at the time the variable is declared. This takes the following form:

```
data-type variable_name = constant;
```

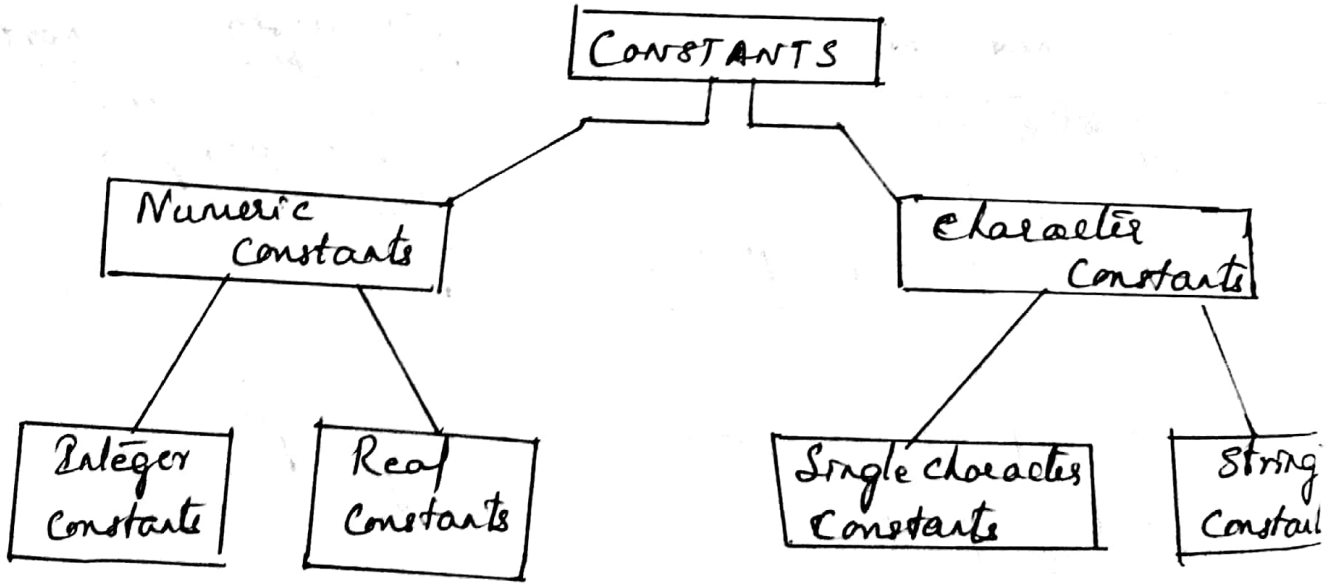
Eg:

```
int final_value = 100;
double balance = 75.84;
```

CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants.

Basic types of C constants



Integer Constants

An integer constant refers to a sequence of digits. There are 3 types of integers, namely, decimal integer, octal integer and hexadecimal integer.

Decimal integers consists of a set of digits, 0 through 9, preceded by an optional – or + sign

valid examples of decimal integer constants are

123, -321, 0, 654321 +78

Reading Data from keyboard

→ A way of giving values is to input data through keyboard using the scanf function. It is the general input function available in C

The general format of scanf is as follows:

scanf ("control string", &variable1, &variables...)

Eg: scanf("%d", &number);

Declaring a variable as a constant

Declaring a variable with the qualifier const at the time of initialization. For example,

const int class_size = 40;

It tells the compiler that the value of the int variable class_size must not be modified by the program.

OPERATORS AND EXPRESSIONS

C supports a rich set of builtin operators. An Operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions.

C operators can be classified into a number of categories. They include:

1. Arithmatic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators.
8. Special operators.

An expression is a sequence of operands and operator that reduces to a single value. For example 10+15 is an expression whose value is 25. The value can be any type other than void.

ARITHMETIC OPERATORS

C provides all the basic arithmatic operators

Arithmatic operator

Operator	meaning
+	Addition or unary plus
-	Subtraction or unary minus.
*	Multiplication
/	division.
%	Modulo division

→ The unary minus operator multiplies its single operand by -1.

→ The modulo division operator % cannot be used on floating point data.

Integer Arithmetic :

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value.

If a and b are integers, if a=14 and b=4, then we have the following results.

a - b = 10

a * b = 56

a / b = 3 (decimal part truncated)

a % b = 2 (remainder of division)

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation.

→ The operator % cannot be used with real operands.

x = 6.0/7.0 = 0.8571

y = 1.0/3.0 = 0.3333

z = -2.0/3.0 = -0.66667

Mixed - mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus 15/10.0 = 1.5 whereas 15/10 = 1.

RELATIONAL OPERATORS

The comparisons can be done with the help of relational operators. An expression such as $a < b$ or $1 < 20$ containing a relational operator is termed as a relational expression. The value of a relational operator expression is either one or zero. It is one if the specified relation is true and zero if the relation is false. For example, $10 < 20$ is true but $20 < 10$ is false.

Relational operators

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal
==	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and takes the following form:

ae-1 relational operator ae-2

where ae-1 and ae-2 are arithmetic expressions which may be simple constants, variable or combination of them. Examples:

4.5 <= 10 TRUE

4.5 <= -10 FALSE

10 < 7+5 TRUE.

Relational operator complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

LOGICAL OPERATORS

C has the following three logical operators:

&&	meaning logical AND
||	meaning logical OR
!	meaning logical NOT

→ The logical operators && and || are used when we want to test more than one condition and make decisions. An example is

```
a > b && x == 10
```

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression.

Truth Table

op-1	op-2	op-1 && op-2	op1 || op2
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Some examples:

1. If (age > 55 && salary < 1000)
2. If (number < 0 || number > 100)

Relative precedence of the relational and logical operators is as follows:

Highest !

> >= < <=

== !=

&&

Lowest ||

ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of a expression to a variable.

→ The usual assignment operator '='.

→ C has a set of 'shorthand' assignment operators of the form

V op = exp ;

where V is a variable, exp is an expression and op is a C binary arithematic operator.

→ The operator op= is known as the shorthand assignment operator.

→ The assignment statement

V op= exp;

is equivalent to

V = V op (exp) ;

with V evaluated only once.

Consider an example

```
x += y+1;
```

This is same the statement

```
x = x+(y+1);
```

→ The shorthand operator += means 'add y+1 to x' or increment 'x' by y+1'

Shorthand Assignment operators

Statement with simple assignment operator	Statement with shorthand operator
a = a+1	a += 1
a = a-1	a -= 1
a = a *(n+1)	a *= n+1
a = a/(n+1)	a /= n+1
a = a % b	a %= b

Advantages of use of shorthand assignment operators

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

CONDITIONAL OPERATOR

A ternary operator "?:" is available in C to construct conditional expressions of the form.

```
exp1 ? exp2 : exp3
```

where exp1, exp2 and exp3 are expressions.

The operator ?: works as follows : exp1 is evaluated first . If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression If exp1 is false, exp3 is evaluated and its value becomes the value of the expression . Note that only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statement

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example , x will be assigned the value of b. This can be achieved using the if --- else statements as follows :

```
If (a>b)
    x=a;
else
    x=b;
```

INCREMENT AND DECREMENT OPERATOR

→ Increment operator ++

→ Decrement operator --

The operator ++ adds 1 to the operand , while -- Substracts 1.

→ Both are unary operators and Takes the following form

```
++m; or m++;
--m; or m--;
```

++m ; is equivalent to m=m+1; (or m+=1;)

--m ; is equivalent to m=m-1; (or m-=1;)

→ We use the increment and decrement statements in for and while loops extensively.

→ While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right hand side of an assignment statement.

Consider the following:

```
m = 5;
y = ++m;
```

in this case the value of y and m would be 6.

If we rewrite the above statements as:

```
m = 5;
y = m++;
```

then the value of y would be 5 and m would be 6.

→ A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

Eg: a[i++] = 10;

is equivalent to

```
a[i] = 10;
i = i+1;
```

Rules for ++ and -- operators :

1. Increment and decrement operators are unary operato and they require variable as their operands.
2. When postfix ++ (or --) is used with a variable in a expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
3. When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
4. The precedence and associativity of ++ and -- operat are the same as those of unary + and unary -.

BITWISE OPERATORS

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level.

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise Exclusive OR
<<	shift left.
>>	shift right.

* The Comma Operator.

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression.

For example, the statement

```
value = (x=10, y=5, x+y);
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (i.e, 10+5) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

* The sizeof operator

The sizeof is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable a constant or a data type qualifier.

Examples:

```
m = sizeof(sum);
n = sizeof(long int);
k = sizeof(235L);
```

The sizeof operator is normally used to determine the length of arrays and structures when their sizes are not known to the programmer.

Pgm to illustrate arithmatic operators

```
main()
{
    int a, b, c, d;
    a = 15;
    b = 10;
    c = ++a - b;
    printf("a = %d  b = %d  c = %d \n", a, b, c);
    d = b++ + a;
    printf("a = %d  b = %d  d = %d \n", a, b, d);
    Printf("a/b = %d \n", a/b);
    printf("a %% b = %d \n", a%b);
    printf("a *= b = %d \n", a*=b);
    printf("%d \n", (c>d)? 1 : 0);
    printf("%d \n", (c<d)? 1 : 0);
}
```

o/p :

```
a = 16   b = 10   c = 6
a = 16   b = 11   d = 26
a/b = 1
a % b = 5
a *= b = 176
0
1
```

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language.

Eg: a*b-c , (m+n)*(x+y) , a*b/c

EVALUATION OF EXPRESSIONS

Expressions are evaluated using an assignment statement of the form:

Variable = expression;

illustration of evaluation of expressions:

```
main()
{
    float a, b, c, x, y, z;
    a = 9;
    b = 12;
    c = 3;

    x = a - b/3 + c * 2 - 1;
    y = a - b/(3+c) * (2-1);
    z = a - (b/(3+c) * 2) - 1;
    printf(" x = %f \n", x);
    printf(" y = %f \n", y);
    printf (" z = %f \n", z);
}
```

```
x = 10.000000
y = 7.000000
z = 4.000000
```

PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are 2 distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + −

Rules for evaluation of expression

- First, parenthesised sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied to determine the order of application of operators in evaluating sub expressions.
- The associativity rule is applied when 2 or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

Mathematical Function

math functions

Function	Meaning
Trigonometric	
acos(x)	Arc cosine of x.
asin(x)	Arc sine of x.
atan (x)	Arc tangent of x.
atan 2(x, y)	Arc tangent of x/y.
cos(x)	Cosine of x.
sin(x)	Sine of x.
tan (x)	Tangent of x.
Hyperbolic	
cosh(x)	Hyperbolic cosine of x.
sinh(x)	Hyperbolic sine of x.
tanh(x)	Hyperbolic tangent of x.
Other functions	
ceil(x)	x rounded upto the nearest integer.
exp(x)	e to the x power (e^x).
fabs(x)	Absolute value of x.
floor (x)	x rounded down to the nearest integer
fmod (x, y)	Remainder of x/y.
log(x)	Natural log of x, x>0.
log10(x)	Base 10 log of x, x>0
pow(x, y)	x to the power y (x^y)
sqrt (x)	Square root of x, x>=0.

Note : 1. x and y should be declared as double.

2. In trigonometric and hyperbolic functions x and y are in radians.

Pgm for quadratic equation

```
#include <math.h>
main()
{
    float a, b, c, dis, root1, root2;
    printf("input the values of a, b & c \n");
    scanf("%f %f %f", &a, &b, &c);
    dis = b*b - 4*a*c;
    if (dis < 0)
     printf("Roots are imagnary \n");
    else
     {
        root1 = (-b + sqrt(dis)) / (2.0*a);
        root2 = (-b - sqrt(dis)) / (2.0*a);
        printf("\n Root1 = %5.2f \n Root2 = %5.2f \n",
                                root1, root2);
     }
}
```

O/P

Input values of a, b, and c

2 4 -16

Root1 = 2.00

Root2 = -4.00

Input values of a, b, and c

(2 3

Roots are imaginary.

Decision making and branching:

C language possesses decision-making capabilities by supporting the following statements:

1) **if** statement.
2) **switch** statement
3) Conditional operator statement.
4) goto statement.

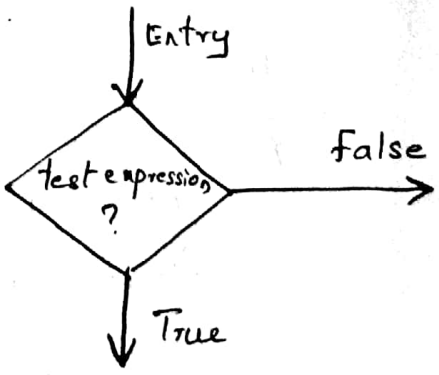
These statements are popularly known as decision-making statements. Since these statements control the flow of execution, they are also known as control statements.

Decision making with if Statement

The **if** statement is a powerful decision-making statement and is used to control the flow of execution of statements. It takes the following form:

```
if (test expression)
```

It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or non-zero) or 'false' (zero), it transfers the control to a particular statement.



Two-way branching

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are:

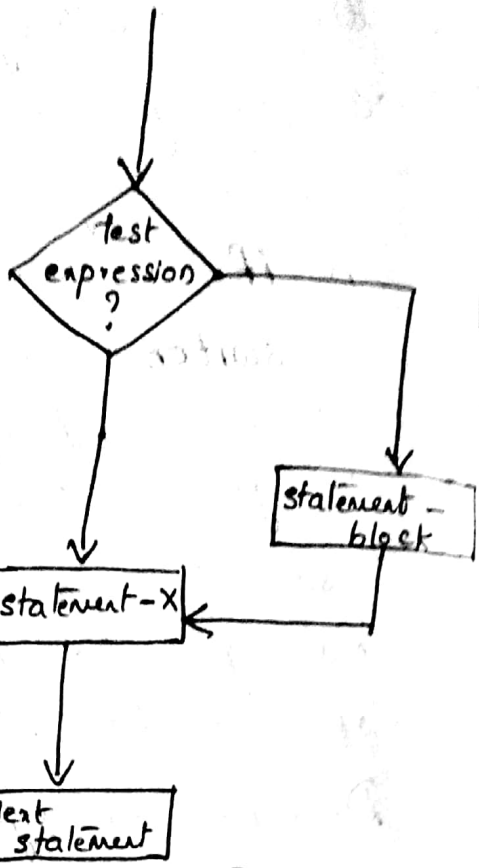
1) Simple if statement.
2) if... else statement.
3) Nested if... else statement.
4) else if ladder.

1) SIMPLE IF STATEMENT

The general form of a simple if statement is

```
if (test expression)
{
    statement-block;
}
statement-x;
```

eg:
```
if (category == sports)
{
  marks = marks + bonus_mark;
}
printf("%f", marks);
```



Flowchart of simple if control

Program-1

The program reads 4 values a, b, c and d from the terminal and evaluates the ratio of (a+b) to (c-d) and prints the result, if c-d is not equal to zero.

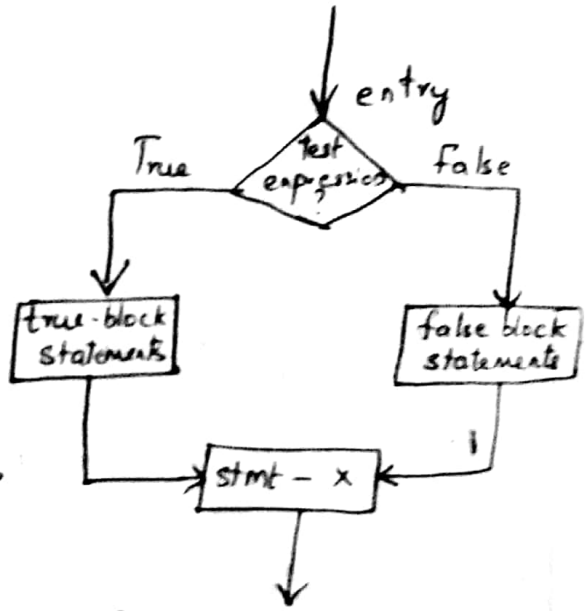
```
#include <stdio.h>
#include <conio.h>

    main()
    {
        int   a, b, c, d;
        float ratio    ;

        printf ("Enter four integer values \n");
        scanf("%d %d %d %d", &a, &b, &c, &d);

        If (c-d != 0)
        {
```

```
        ratio = (float)(a+b)/(float)(c-d);
        printf("Ratio = %f \n", ratio);
    }
  getch();
}
```

(2) THE IF... ELSE STATEMENT

The if-else statement is an extension of the simple if statement.
The general form is:

```
if (test expression)
    {
      true-block statements;
    }
else
    {
      false-block statements;
    }
```



flow chart of if...else control

If the test expression is true then the true-block statements otherwise false-block statements (s) are executed.

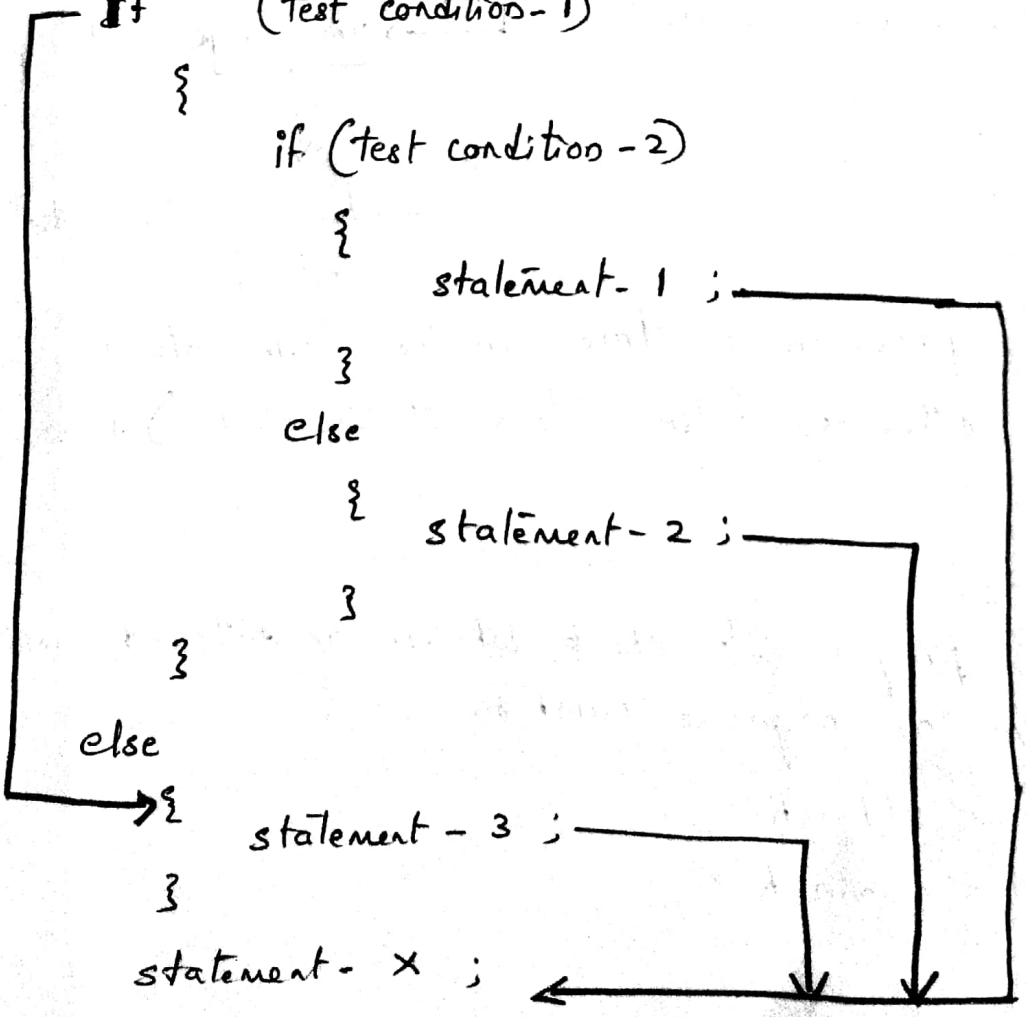
Program-2

A sample program to check whether an entered number is positive or negative number.

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

```
void main()
{
    int a ;
    printf("Enter a non-zero number \n");
    Scanf("%d", &a);
    If (a >= 0)
       printf(" Positive number \n");
    else
       printf(" Negative number \n");
  getch();
}
```

(3) NESTING OF IF.... ELSE STATEMENTS



Program-3

Selecting the largest of 3 numbers

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a, b, c;
   printf("Enter three values\n");
   scanf("%d%d%d", &a, &b, &c);
   printf("Largest value is ");
   if (a > b)
      {
         if (a > c)
            printf("%d\n", a);
         else
            printf("%d\n", c);
      }
   else
      {
         if (c > b)
            printf("%d\n", c);
         else
            printf("%d\n", b);
      }
   getch();
}
```

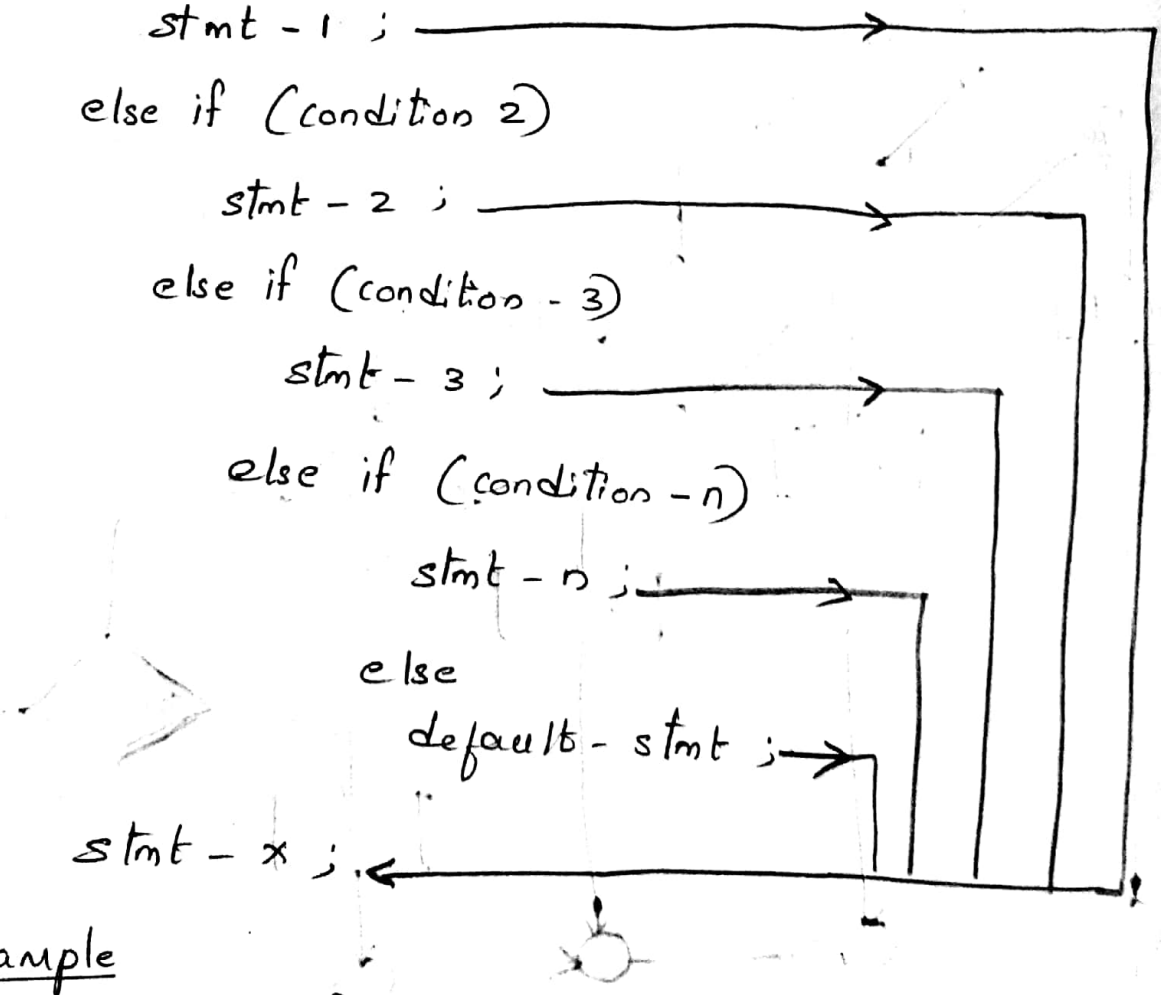
Dangling else problem

This occurs when a matching else is not available for an if. The answer to this problem is always match an else to the most recent unmatched if in the current block.

(4) THE ELSE IF LADDER

It takes the general form:

```
if (condition 1)
    stmt-1 ;
else if (condition 2)
    stmt-2 ;
else if (condition-3)
    stmt-3 ;
else if (condition-n)
    stmt-n ;
else
    default-stmt ;
stmt-x ;
```

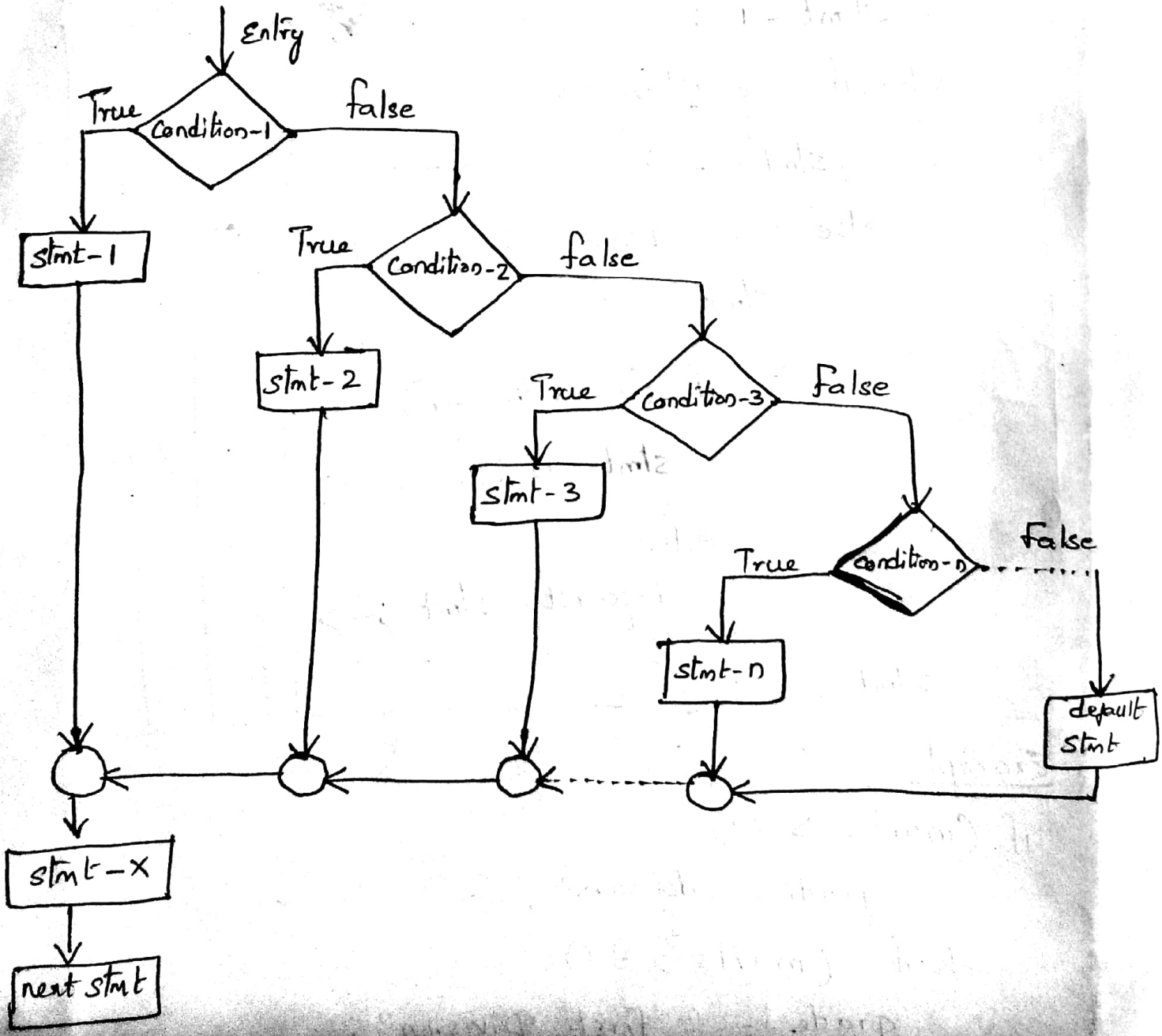


Example

```
if (marks > 79)
    grade = "Honours";
elseif (marks > 59)
    grade = "first Division";
else if (marks > 49)
    grade = "second Division";
```

```
else if (marks > 39)
        grade = "Third Division"
    else
grade = "Fail";
printf("%s\n", grade);
```

Flow chart of else... if ladder



SWITCH STATEMENT

C has built-in multiway decision statement known as switch. The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:

```
Switch (expression)
{
    case value-1 :
                block-1
                break;
    Case value-2 ;
                block-2
                break;
    .....
    ......
    default:
                default block
                break;
}
statement-x;
```

→ The expression is an integer expression or characters. value-1, value-2 .... are constants or constant expressions and are known as case labels.

→ Each of these values should be unique within a switch statement. block-1, block-2 are statement lists and may contain zero or more statements.

→ There is no need to put braces around these blocks.

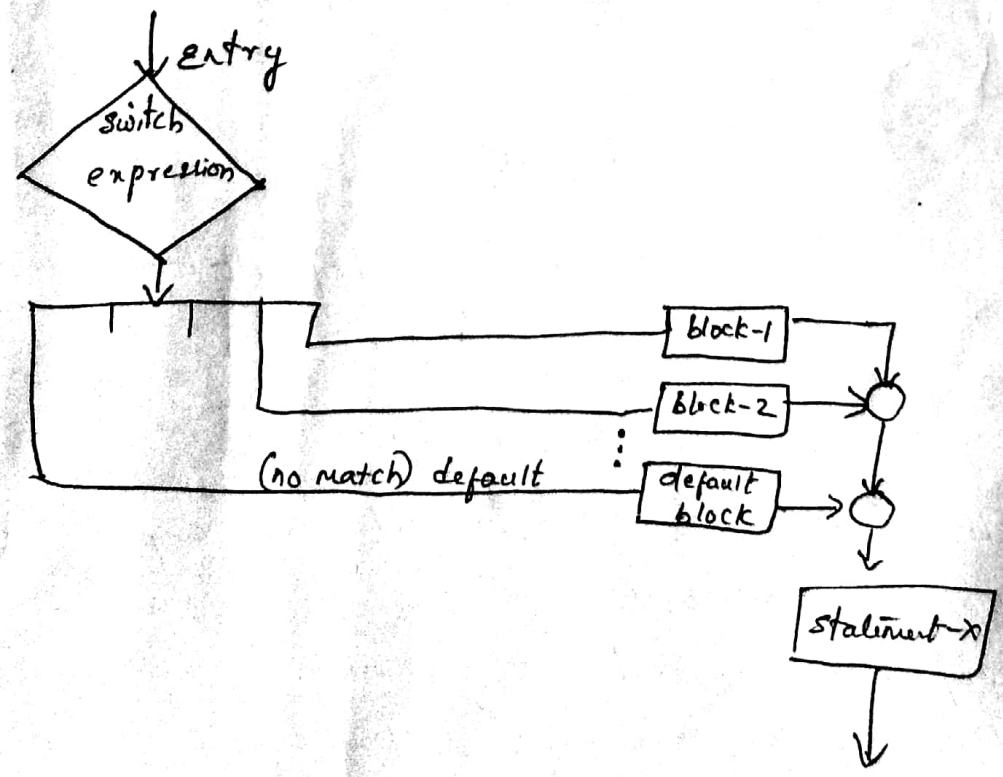
Note that case labels end with a colon (:).

When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2, ... If a case is found whose value matches with the value of the expression, the block of statements that follow the case are executed.

The break statement at the end of block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-x following the switch.

→ The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place, if all matches fail and the control goes to the statement-x.

→ The switch expression must be an integral type.



Selection process of the switch statement

The GOTO statement

C supports the goto statement to branch unconditionally from one point to another in the program.

→ The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of goto and label statements are shown below;

forward Jump

```
goto label;
.......
.......
.......
label:
  statement;
```

Backward Jump

```
label:
 statement;
.......
.......
.......
goto label;
```

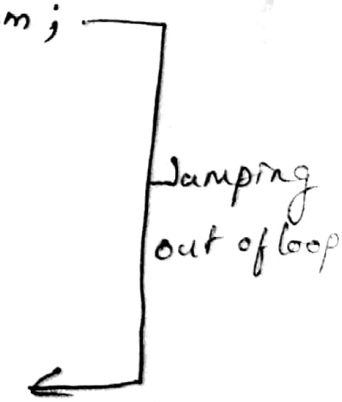
Eg:-

```
main()
{
   float x, y;
   read:
   scanf("%f", &x);
   if (x < 0) goto read;
   y = sqrt(x);
   printf("%f %f", x, y);
}
```

Another use of the goto statement is to transfer the control out of a loop when certain conditions are encountered.

Eg:

```
- - - - - -
- - - - - - -
while(. . . .)
 {
    for(. . . .)
    {
      - - - - -
      - - - - -
      if(. . . .) goto end_of_pgm;
      - - -
    }
    - - - -
    - - - -
 }
end_of_program:
```



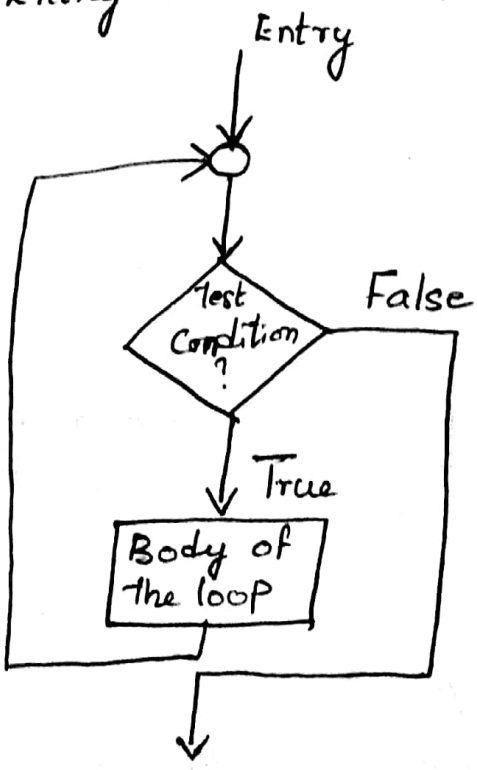
Decision making and Looping:

In looping a sequence of statements are executed until some condition for the termination of the loop is satisfied.

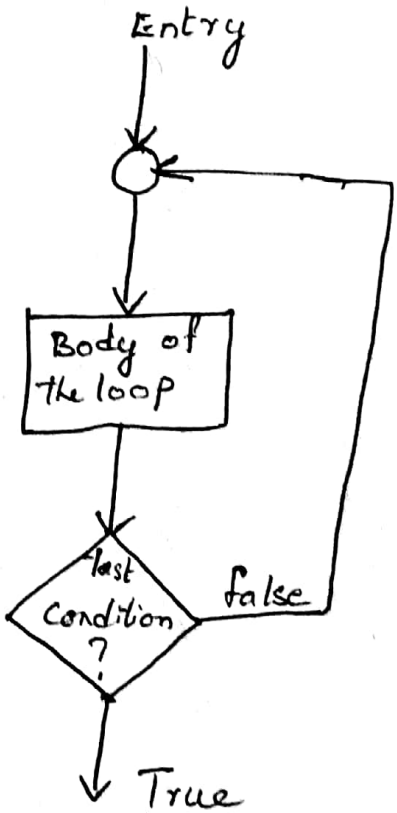
Decision making and looping.

In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied. A program loop therefore consists of 2 segments, one known as the body of the loop and other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop. Depending on the position of the control statement in the loop, a control structure may be classified either as the entry controlled loop or as the exit-controlled loop.

a) Entry controlled loop



b) Exit controlled loop.



In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop

will not be executed. In case of an exit controlled loop the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled and exit-controlled loops are also known as pretest and post test loops respectively.

A looping process, in general, would include the following four steps:

1. setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

The C language provides for three constructs for performing loop operations. They are:

1. The while statement.
2. The do statement.
3. The for statement.

Sentinel loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops may be classified into 2 general categories:

1. Counter-controlled loops.
2. Sentinel-controlled loops.

When we know in advance exactly how many times a loop will be executed, we use a counter-controlled loop. We use a control variable known as counter. The counter must be initialised, Tested and updated properly for

the desired loop operations. The number of times we want to execute the loop may be a constant or a variable that is assigned a value. 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. For example, when reading data we may indicate the "end of data" by a special value like -1 and 999. The control variable is called sentinel variable. A sentinel-controlled loop is often called indefinite repetition loop because the number of repetitions is not known before loop begins executing.

THE WHILE STATEMENT

The basic format of the while statement is

```
while (test condition)
{
   body of the loop;
}
```

The while is an entry-controlled loop statement. The test-condition is evaluated and if the condition is true, then the body of the loop is executed. After the execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop. On exit, the program

continues with the statement immediately after the body of the loop.

```
Eg:   Sum = 0;
      n = 1;  - - - - - - - - - -> /* initialisation */
      while (n <= 10) - - - - - -> /* testing */
      {
          Sum = Sum + n * n;
          n = n + 1;  ---------> /* incrementing */
      }
      printf ("sum = %d", sum);
      - - - - - -
      - - - - -
```

loop

The variable n is called counter or control variable.

THE DO STATEMENT

The while loop construct that we have discussed makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. The do statement takes the form:

```
do
{
    body of the loop
} while (test-condition);
```

On reaching the do statement, the program proceeds to evaluate the body of the loop first. At the end of loop, the test-condition in the while statement is eval

If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediatly after the while statement.

Since the test-condition is evaluated at the bottom of the loop, the do...while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

Eg:

```
sum = 0;
do
{
  sum = sum + i;
  i = i + 2;
} while (sum < 40 || i < 10);
printf ("%d %d \n", i, sum);
```

THE FOR STATEMENT

The for loop is another entry-controlled loop that provides a more concise loop control structures. The general form of the for loop is:

```
for (initialisation ; testcondition ; increment)
{
  body of the loop;
}
```

→ Example for "for loop"

```
for (x=0; x<=9; x++)
    {
     printf("%d ", x);
    }
printf("\n");
```

The for loop is executed 10 times and prints the digit 0 to 9 in one line.

Eg:2

```
for (x=9; x>=0; x=x-1)
    printf("%d", x);
printf("\n");
```

The loop is executed 10 times but the output would be from 9 to 0.

Comparison of the three loops

for

```
for(n=1; n<=10;
      ++n)
{
 ____________
 ____________
}
```

while

```
n=1;
while(n<=10)
{
 ____________
 ____________
n=n+1;
}
```

do

```
n=1;
do
{
 ____________
 ____________
n=n+1;
}while(n<=10);
```

Nesting of for loops

Nesting of loops, that is, one "for" statement within another "for" statement.

Example:

```
for (i=1 ; i<10 ; i++)
  {
  ---------
  ---------
  for (j=1 ; j!=5 ; ++j)
    {
    ---------
    ---------
    }
  ---------
  ---------
  }
```

inner loop

outer loop.

Pgm to print the powers of 2 table for the power of to 20. both positive and negative [$P = 2^n$ & $q = 2^{-n} = \frac{1}{P}$]

```
void main()
{
    long int P;
    int n;
    double q;
    printf("-------------------------\n");
    printf("2 to power n    n    2 to power -n \n");
    printf("-------------------------\n");
    P=1;
```

```
for (n=0; n<21; ++n)
{
    if (n==0)
        p=1;
    else
        p= p*2;
    q = 1.0/(double)p;
    printf ("%d    %d    %5.12lf \n", p, n, q);
}
printf ("-----------------------\n");
}
```

output :

2 to power n	n	2 to power - n
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
.	.	.
.	.	.
.	.	.
1048576	20	0.000009536743

JUMPS IN LOOPS

* Jumping out of a loop:

An early exit from a loop can be accomplished by using the break statement or the goto statement.

when a break statement is encountered inside a loop the loop is immediatly exited and the program continues with the statement immediatly following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the break will exit only a single loop.

A goto statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of goto is to exit from deeply nested loops when an error occurs.

Eg: ①

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

Exit form loop

②

```
do
{
.....
.....
if(condition)
    break;
.....
.....
}while(....);
.....
```

Exit from loop

③

```
for (- - - - -)
{
    - - - - -
    - - - - -
    if (error)
        break;
    - - - - -
    - - - - -
}
- - - - - -
```

Exit from loop

④

```
for (- - - - -)
{
    - - - - -
    for (- - - - -)
    {
        - - - - -
        if (condition)
            break;
        - - - - -
    }
    - - - - - -
}
```

Exit from inner loop

Jumping within and exiting from the loops with goto statement

a)

```
while (- - - - -)
{
    if (error)
        goto stop;
    - - - - -
    if (condition)
        goto abc;
    - - - - -
    - - - - -
abc:
    - - - - -
}
stop:
- - - - -
- - - - -
```

Jump within loop

b)

```
for (- - - - -)
{
    - - - - -
    for (- - - - -)
    {
        - - - - -
        if (error)
            goto error;
        - - - - -
    }
    - - - - -
}
error:
- - - - -
- - - - -
```

Exit from 2 loops

skipping a part of a loop

C supports another statement called the continue statement. The continue statement causes the loop to be continued with the next iteration after skipping the statements in between. The continue statement tells the compiler, "skip the following statements and continue with the next iteration". The format of the continue statement is simply

```
continue ;
```

eg:

a)
```
while(test condition)
{
    ----
    if (----)
      continue;
    ----
    ----
}
```

b)
```
do
{
    ----
    if(----)
      continue;
    ----
    ----
} while (test condition);
```

c)
```
for (initialisation ; test condition ; increment)
{
    ----
    if(----)
      continue;
    ----
    ----
}
```

Jumping out of the program

Jump out of a program by using the library function exit(). The use of exit() function requires the header file <stdlib.h>

Difference btn while loop & do while loop.

While loop

1. Entry-controlled loop(checks whether the condition specified is true before executing the stmts in the body of the loop.
2. Doesn't execute even once if the condition is false.
3. It is generally more useful and more commonly used.
4. No semicolon at the end of the loop syntax.
5. Syntax:

```
while (i<=0)
{  i = i/10;
}
```

Do while loop

1. Exit-controlled loop (checks the condition after executing the body of the loop)
2. Executes at least once, even if the condition is false
3. Generally less useful and much less commonly used, as compared to while loop.
4. Semicolon is required at the end of loop syntax.
5. Syntax:

```
do {
   i = i/10;
} while (i<=0);
```

Reading a character

* Reading a single character can be done by using the function getchar.
* The getchar takes the following form:

```
variable_name = getchar();
```

variable_name is a valid C name that has been declared as character type.

Eg:
```
char name;
name = getchar();
```

pgm-1:

```
#include <stdio.h>
main()
{
    char answer;
    printf("would you like to know my name?\n");
    printf("Type Y for yes and N for No\n");
    answer = getchar();
    if (answer == 'Y' || answer == 'y')
    printf("\n my name is Busy Bee\n");
    else
     printf("\n You are good for nothing\n");
}
```

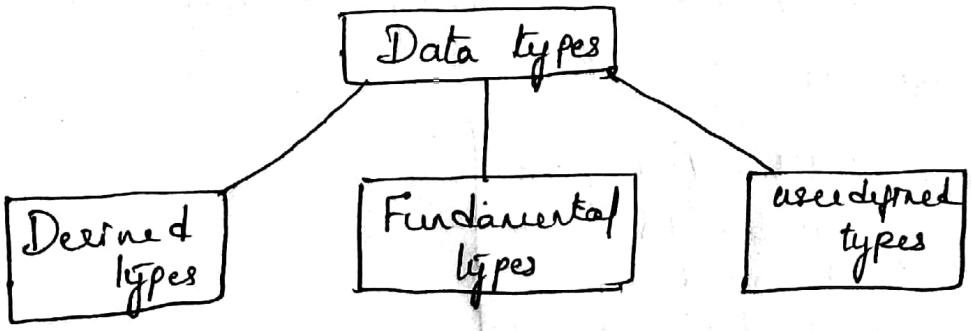
MODULE - 3

ARRAYS

An array is a fixed-size sequenced collection of elements of the same data type. It is simply a grouping of like-type data. In its simplest form, an array can be used to represent a list of numbers or a list of names. Since an array provides a convenient structure for representing data, it is classified as one of the data structures in C. There are different types of arrays:

1. One dimensional array.
2. Two dimensional array.
3. Multi dimensional array.

Data structures



- Arrays
- Functions
- pointers

- Integral
- Float
- character

- structures
- Union
- Enumerations

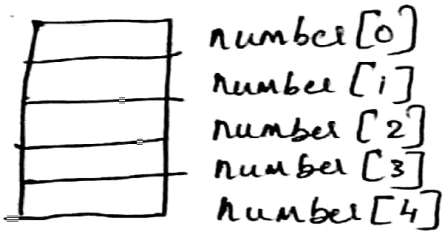
ONE - DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a

one dimensional array.

For example: If we want to represent a set of five numbers say (35, 40, 20, 57, 19) by a variable number, then we may declare the variable number as follows:

int number[5]; and the computer reserves 5 storage locations as shown below:



The values to the array elements can be assigned as follows:

number[0] = 35 ; number[1] = 40 ; number[2] = 20 ;
number[3] = 57 ; number[4] = 19 ;

This would cause the array number to store the values as shown below.

number[0]	35
number[1]	40
number[2]	20
number[3]	57
number[4]	19

DECLARATION OF ONE DIMENSIONAL ARRAY

The general form is,

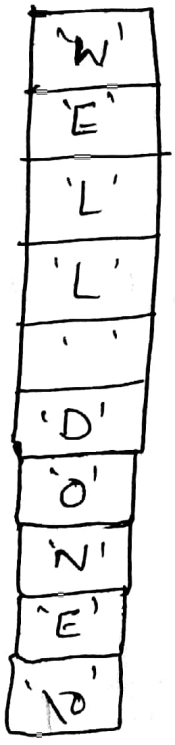
type variable_name [size];

The type specifies the type of element that will be contained in the array such as int, float or char

and the size indicates the maximum number of elements that can be stored inside the array.

For eg: float height[50]; declare the height to be an array containing 50 real elements.

Eg: char name[10]; declares the name as a character string variable that can hold a maximum of 10 characters. Suppose "WELL DONE". Each character of the string is treated as an element of the array name and is stored in the memory as follows:



'\0' hold the null character. When declaring the character arrays, we must declare one extra element space for the null terminator.

INITIALISATION OF ONE-DIMENSIONAL ARRAY

After an array is declared, its elements must be initialised. Other wise they will contain garbage.

An array can be initialised at either of the follo. stages:

1) At compile time.
2) At run time.

Compile time initialisation:

The general form of initialization of arrays is:

type array-name [size] = { list of values };

Eg: the statement

int a[3] = {0, 0, 0}

will declare the variable ~~numbers~~ a as an array of size 3 and will assign zero to each element. The size may be omitted. In such cases, the compiles allocates enough space for all initialised elements. for eg:

int c[] = {1, 1, 1, 1};

will declare the c array to contain 4 elements with initial values 1.

The character arrays may be initialised in a similar manner.

char name[] = {'J', 'o', 'h', 'n', '\0'};

declares the name to be an array of 5 characters initialised with the string "John" ending with null character '\0'. Alternatively, we can assign the string literal directly as under

char name[] = "John";

RUN TIME INITIALIZATION:

An array can be explicitly initialized at run time. This approach is applied for initialising large arrays:

for eg:

```
for (i=0; i<100; i++)
        sum[i] = 0;
```

Reading values into an array.

```
int a[10];
for (i=0; i<10; i++)
    scanf("%d", &a[i]);
```

Searching & Sorting

Searching & sorting are the most frequent operations performed on arrays.

TWO-DIMENSIONAL ARRAYS

If a table of values have to be stored then we use two dimensional arrays:

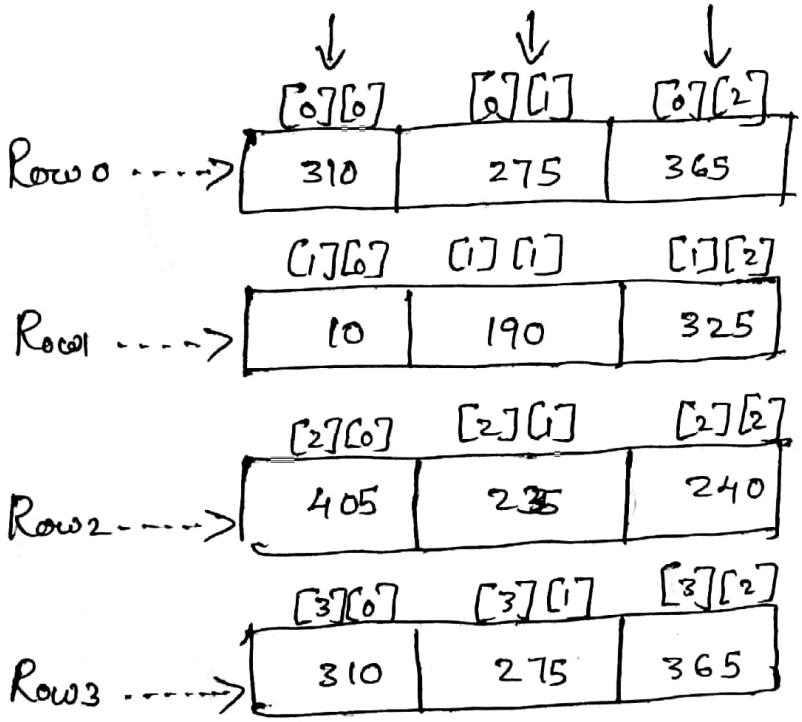
Consider the following data table, which shows the value of sales of 3 items by four Sales girls.

	item1	item2	item-3
Salesgirl #1	310	275	365
Salesgirl #2	210	190	325
Salesgirl #3	405	235	240
Salesgirl #4	260	300	380

The table contains a total of 12 values, three each line. We consider this table as a matrix consisting of 4 rows and 3 colowns. Each row represents the values of sales by a particular sales-girl and each colown represents the values of sales of a particular item.

Two dimensional arrays are declared as follows:

Type array-name [row-size] [colown-size];



Representation of a 2-D array in memory

INITIALIZING TWO-DIMENSIONAL ARRAYS.

Two-dimensional arrays may be initialised by following their declaration with a list of initial values enclosed in braces.

For eg:

```
int table[2][3] = {0, 0, 0, 1, 1, 1};
```

The initialization is done row by row. The above statement can be equivalently written as:

```
int table[2][3] = { {0, 0, 0}, {1, 1, 1} };
```

by surrounding the elements of the each row by braces. We can also initialise a 2D array in the form of a matrix as shown below:

```
int table[2][3] = {
                    {0, 0, 0},
                    {1, 1, 1}
                  };
```

Eg:

```
int table[][3] = {
                    {0, 0, 0},
                    {1, 1, 1}
                  };
```

When all the elements are to be initialised to zero, the following short-cut method may be used:

```
int m[3][5] = { {0}, {0}, {0} };
```

The first element of each row is explicitly initialised to zero while others elements are automatically initialised to zero. The following statement will also achieve the same result:

```
int a[3][5] = {0, 0};
```

Memory Layout:

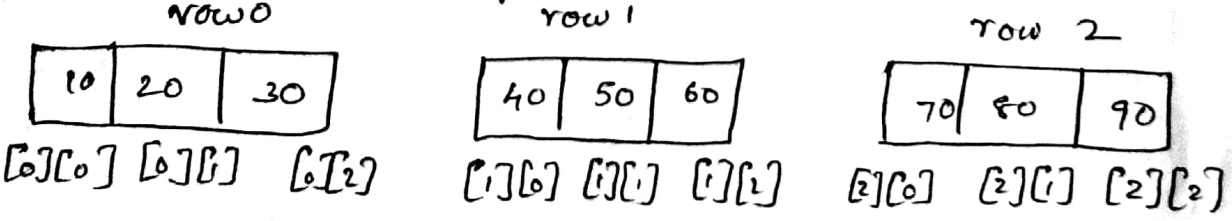
The subscripts in the definition of a two-dimen. array represent rows and colowns. This format maps the way that the elements are laid out in the memory. The ~~elements~~ elements of all arrays are stored contiguously in increasing memory locations, essentially in a single list. If we consider the memory as a row of bytes, with the lowest address on the left and the highest address on the right, a simple array will be stored in the memory with the first element at the left end and the last element at the right end. Similarly, a 2-D array is stored "row-wise, starting from the first row and ending with the last row, treating each row like a simple array. This is illustrated below.

colows

row	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

3 × 3 array.

Memory Layout:



READING AND PRINTING ELEMENTS INTO A 2-D ARRAY

Reading:

```
for(i=0; i<m; i++)
    for(j=0; j<n; j++)
        Scanf("%d", &a[i][j]);
```

printing is done by:

```
for (i=0 ; i<m; i++)
    {
      for (j=0; j<n ;j++)
          {
            print (" %d\t", a[i][j]);
          }
      printf ("\n");
    }
```

MULTIDIMENSIONAL ARRAYS:

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is:

Type array-name $[s_1][s_2][s_3]\ldots[s_n]$;

where s_i is the size of the i^{th} dimension. some examples are:

```
int survey [3][5][12];
float table [5][4][5][3];
```

Dynamic arrays:

An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run-time. The process of allocating memory at compile time is known as static memory allocation, and the arrays that receive static memory allocation are called static arrays.

Dynamic arrays are created using what are known as pointer variables and memory management functions like malloc, calloc and realloc. These functions are included in the header file <stdlib.h>.

CHARACTER ARRAYS AND STRINGS

A string is a sequence of characters that is treated as a single data item. Any group of characters (except double quote sign) defined b/t double quotation marks is a string constant.

For eg:

"Man is obviously made to think"

Declaring and initializing string variables.

C does not support strings as a data type. However, it allows us to represent strings as character arrays. In C, therefore, a string variable is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is,

```
char string_name[size];
```

The size determines the number of characters in the string_name. Some examples are:

```
char city[10];
char name[30];
```

When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

Reading a ~~line~~ ~~of~~ Text Using getchar and gets function:

→ Read a single character from the terminal is given by

```
char ch;
ch = getchar();
```

→ Pgm for reading a line of text from terminal

```
main()
{ char line[81], c;
  int c1 = 0;
  printf("enter the text. press <Return> to
          end \n");
  do
  { c = getchar();
    line[c1] = c;
    c1++;
  } while (c != '\n');
  c1 = c1 - 1;
  line[c1] = '\0';
  printf("\n %s \n", line);
}
```

```
s[i]=='i' || s[i]=='o' || s[i]=='u')
    vowels++;
  else
    consonants++;
}
printf(" %d vowels and %d consonants are present",
                    vowels, consonants);
}
```

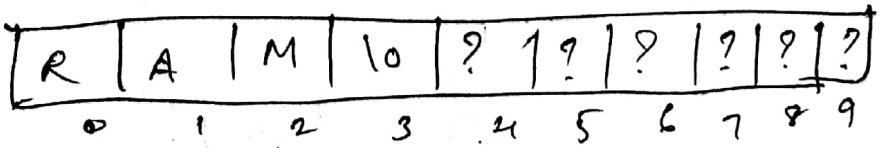
→ We can also specify the field width using the form %ws in the scanf statement for reading specified number of characters from the input string.

For example: scanf("%ws", name)

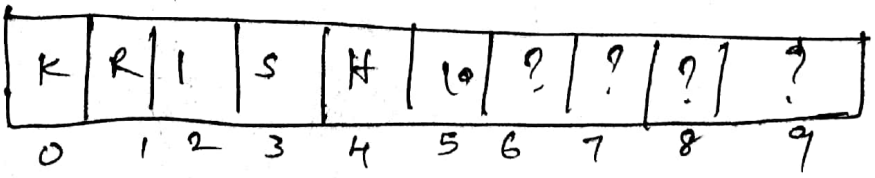
eg:

```
char name[10];
scanf("%5s", name);
```

The input string RAM will be stored as:



The input string KRISHNA will be stored as



ie ampersand (&) is not required before the variable name.

```
#include<stdio.h>
main()
{ char str[20] = " Hello World";
  printf("%s\n", &str);
}
```

* Pgm to find the length of the string.

```
main()
{ char s1[] = "TajMahal";
    int i=0;
  while(s1[i] != '\0')
    { i = i+1;
    }
printf("The length of the string is %d", i);
}
```

* Pgm to find the no: of vowels and consonants in a string.

```
main()
{ char s[] = "Tajmahal";
  int i=0, vowels=0; consonants=0;
  while (s[i++] != '\0')
    { if (s[i] == 'a' || s[i] == 'e'
```

C permits a character array to be initialized in either of the following 2 forms:

```
char city[9] = "New York";
char city[9] = {'N', 'E', 'W', ' ', 'Y', 'O', 'R', 'K', '\0'};
```

C also permits us to initialise a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialised.

```
char string[] = {'G', 'O', 'O', 'D', '\0'};
```

Reading strings from terminal:

(a) Using scanf:

Input function scanf can be used with %s format specification to read in a string of characters.

```
char address[20];
scanf("%s", address);
```

The problem with the scanf function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage-returns, formfeeds and new lines.

gets

Another method of reading a string of text containing whitespaces is to use the library function gets available in the <stdio.h> header file. This is a simple function with one string parameter and called as under:

```
gets (str);
```

str is a string variable declared properly. It reads characters into str from the keyboard until a new-line character is encountered and then appends a null character to the string.

For eg: char line[80];

```
gets (line);
printf ("%s", line);
```

Writing strings to screen:

Using printf function:

The printf function with %s format to print strings to the screen. The format is...

printf ("%s", name); can be used to display the entire contents of the array name.

Using putchar and puts functions

C supports another character handling functions

putchar to output the values of character variables. It takes the following form:

```
char ch = 'A';
putchar(ch);
```

The function putchar requires one parameter. The statement is equivalent to:

```
printf("%c", ch);
```

Another way of printing string values is to use the function puts declared in the header file <stdio.h>. This is one parameter function and invoked as under:

```
puts(str);
```

For eg:-

```
char line[80];
gets(line);
puts(line);
```

STRING - HANDLING FUNCTIONS

It is included in <string.h>

	Function	Action
1.	strcat()	concatenates 2 strings.
2.	strcmp()	Compares 2 strings.
3.	strcpy()	copies one string over another
4.	strlen()	finds the length of a string.

1. strcat() function

The strcat function joins 2 string together. it takes the following form:

strcat (string1, string2);

string1 and string2 are character arrays. when the function strcat is executed, string2 is appended to the string. It does so by removing the null character at the end of string1 and placing string2 from there.

c permits nesting of strcat functions. For eg the stmt.

strcat (strcat(string1, string2), string3);

The result is stored in string1.

2. strcmp() function

The strcmp function compares 2 strings identified by the arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first non matching characters in the strings. It takes the following form)

strcmp (string1, string2);

string1 and string2 may be string variables or string constants.

Examples are:

```
strcmp(name1, name2);
strcmp ("Ram", "Ram");
```

For eg: the statement strcmp ("their", "there"); will returns a value of -9 which is the numeric difference between ASCII "i" and ASCII "r".

3. strcpy() function

The strcpy function works almost like a string assignment operator. It takes the form:

```
strcpy(string1, string2);
```

and assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

for eg: the statement,

```
strcpy(city, "Delhi");
```

will assign the string "Delhi" to the string variable city. similarly, the statement,

```
strcpy(city1, city2);
```

will assign the contents of the string variable city2 to the string variable city1.

4. strlen() function:

This function counts and returns the number of characters in a string. It takes the form.

```
n = strlen(string);
```

where n is an integer variable which receives the value of the length of the string.

Module - 5

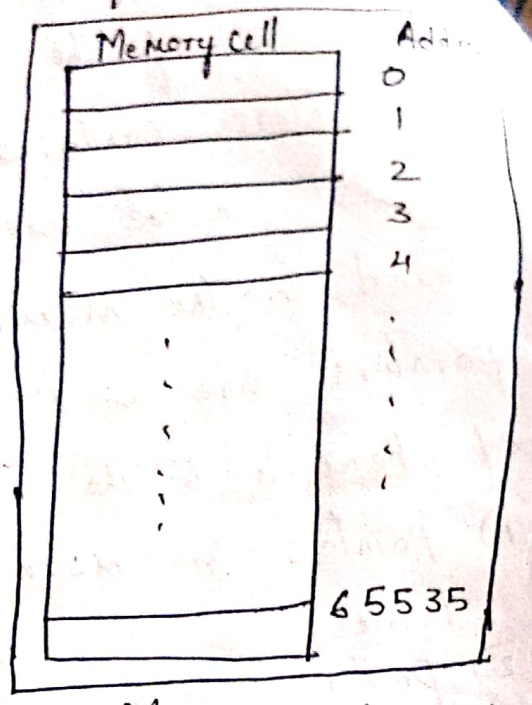
A pointer is a derived data type in C. It is build from one of the fundamental data types available in C. pointers contains memory addresses as their values. pointers can be used to access and manipulate data stored in the memory.

pointers are used frequently in C, as they offer a numbes of benefits to the progoammers. They include:

1) Pointers are more efficient in handling arrays & data tables.
2) Pointers can be used to return multiple values from a function via function arguments.
3) Pointers permit references to functions and thereby facilitating pasing of functions as arguments to other functions.
4) Pointers allow C to support dynamic memory management.
5) The use of pointer arrays to character strings results in saving of data storage space in memory.
6) pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7) pointers reduce the length and complexity of programs.
8) They increase the execution speed and thus reduce the program execution time.

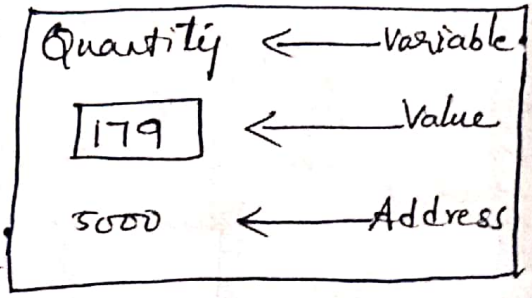
Understanding Pointers

The computer's memory is a sequential collection of storage cells. Each cell is commonly known as a byte, has a number called address associated with it. Typically, the addresses are numbered consecutively starting from zero. The last address depends on the memory size of a computer system. for eg, if it is having 64k memory then it will have its last address as 65535.



Memory Organisation

When we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number.



Consider, the following statement

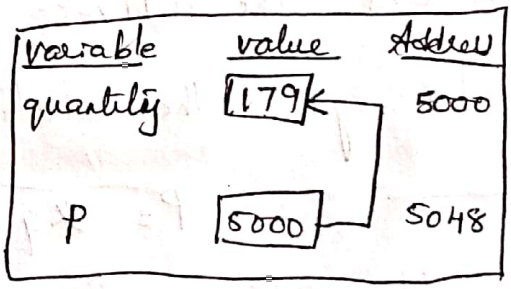
```
int quantity = 179;
```

This statement instructs the system to find a location of the integer variable quantity and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for quantity. We may access the ~~the~~ value 179 by using either the name quantity or the address 5000.

Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables.

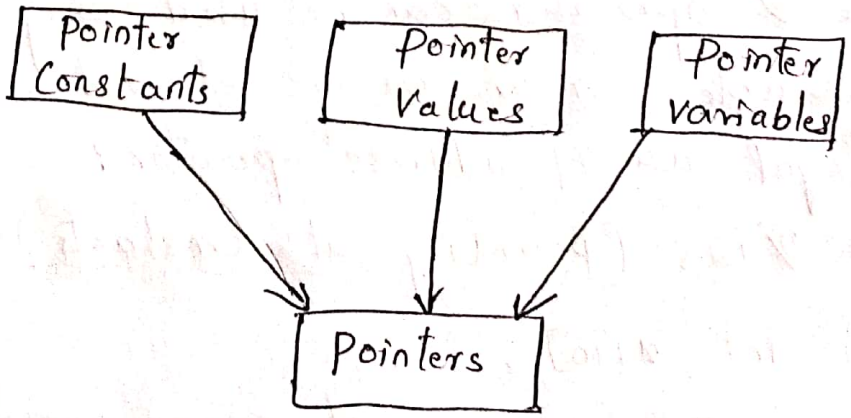
A pointer variable is nothing but a variable that contains an address, which is a location of another variable in memory

Since a pointer is a variable, its value is also stored in the memory in another location. Suppose we assign quantity to a variable p, the link b/w variables p and quantity is given by the figure. The address of p is 5048.



Concepts of Pointers

Pointers are built on the three underlying concepts:



→ Memory addresses within a computer are referred to as pointer constants.

→ We cannot save the value of a memory address directly. We can only obtain the value through the

variable stored there using the address operator. The value thus obtained is known as pointer value.

→ Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a pointer variable.

ACCESSING THE ADDRESS OF A VARIABLE.

→ The actual location of a variable in the memory is system dependent.

→ The operator & immediately preceding a variable returns the address of the variable associated with it.

For eg: the statement

```
P = &quantity
```

would assign the address 5000 (the location of quantity) to the variable p. The & operator can be remembered as 'address of'.

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1) &125 (pointing at constants)

2) int x[10];
   &x (pointing at array names).

3) &(x+y) (pointing at expressions).

if x is an array, then expressions such as

&x[0] and &x[i+3]

are valid and represent the addresses of 0th and (i+3)th elements of x.

A pgm to print the addresses of a variable along with its values.

```
void main()
{
    char a;
    int x;
    float p, q;
    a = 'A';
    x = 125;
    p = 10.25, q = 18.76;
    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);
}
```

o/p

A is stored at addr 4436.

125 is stored at addr 4434.

10.250000 is stored at addr 4442.

18.760000 is stored at addr 4438.

DECLARING POINTER VARIABLES.

The declaration of a pointer variable takes the following form:

data_type *pt_name ;

This tells the compiler three things about the variable pt_name;

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data_type.

for eg:

int *p ; /* integer pointer */

→ declares the variable p as a pointer variable that points to an integer data type.

float *a ; /* float pointer */

→ declares a as a pointer to a floating point variable.

pointer declaration style

1) int *p ;
2) int* p ;
3) int *p ;

INITIALIZATION OF POINTER VARIABLES

The process of assigning the address of a variable to a pointer variable is known as initialization.

For eg:

```
int quantity ;
int *p ;          /* declaration */
p = &quantity ;   /* initialisation */
```

We can also combine the initialization with the declaration. That is.

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

The statement:

```
int x, *p = &x ;   /*three in one*/
```

is perfectly valid.

But the statement:

```
int *p = &x, x ;
```

is not valid.

We could also define a pointer variable with as initial value of NULL or 0 (zero). That is:

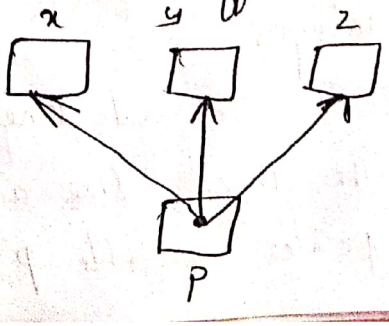
```
int *p = NULL ;
int *p = 0 ;
```

Pointer flexibility:

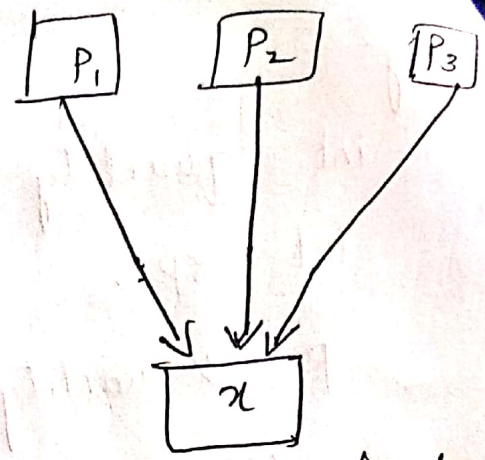
pointers are flexible. We can make the same pointer to point to different data variables in different statements:

Eg:

```
int x, y, z, *p ;
...
p = &x ;
...
p = &y ;
p = &z ;
```



```
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
- - - - -
- - - -
```



We can also use different pointers to point to the same data variable as given above.

ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, how will you access the value of the variable using the pointer? This is done by another unary operator * (asterisk) usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;
```

→ The first line declares quantity and n as integer variables and p as a pointer variable pointing to an integer.

→ The second line assigns the value 179 to quantity.

→ The Third line assigns the address of quantity to the pointer variable p.

The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression, the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable quantity, because p is the address of quantity.

For eg: p = &quantity;
n = *p;
are equivalent to,
n = *(&quantity);

A pgm to illustrate the use of indirection operator * to access the value pointed to by a pointer.

```
void main()
{
    int x, y;
    int *ptr;
    x = 10;
    ptr = &x;
    y = *ptr;
    printf("value of x is %d\n", x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
```

```
printf ("%d is stored at addr %u\n", ptr, &ptr);
printf ("%d is stored at addr %u\n", y, &y);
*ptr = 25;
printf ("\n Now x = %d\n", x);
}
```

o/p

Value of x is 10.

10 is stored at addr 4104.

to is stored at addr 4104

to is stored at addr 4104.

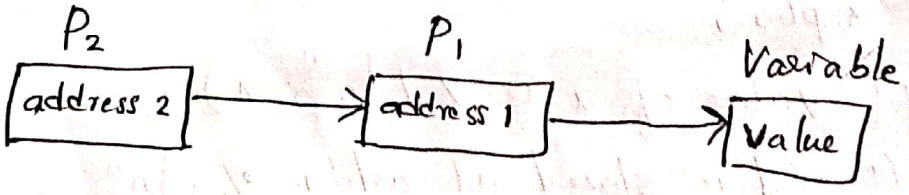
4104 is stored at addr 4106

is stored at addr 4108

Now x = 25

CHAIN OF POINTERS

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown below.



The pointer variable P_2 contains the address of the pointer variable P_1, which points to the location that contains the desired value. This is

known as multiple indirections.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Eg:

int **p2;

This declaration tells the compiler that p_2 is a pointer to a pointer of int type. Remember, the pointer p_2 is not a pointer to an integer, but rather a pointer to an integer pointer.

```
void main()
{
    int x, *p1, **p2;
    x = 100;
    p1 = &x;        /* Address of x */
    p2 = &p1;       /* address of p1 */
    printf("%d", **p2);
}
```

This code will display the value 100. Here p_1 is declared as a pointer to an integer and p_2 as a pointer to a pointer to an integer.

POINTER EXPRESSIONS

Pointer variables can be used in expressions. For eg: If p_1 and p_2 are properly declared & initialised pointers, then

```
y = (*p1) * (*p2);
Sum = Sum + *p1;
Z = 5 * - (*p2)/(*p1);
*p2 = *p2 + 10;
```

We can also use short hand operators with the pointers:

```
p1++;
--p2;
Sum += *p2;
```

A sample program

```
void main()
{   int a, b, *p1, *p2, x, y, z;
    a = 12;
    b = 4;
    p1 = &a;
    p2 = &b;
    x = *p1 * *p2 - 6;
    y = 4 * - *p2 / *p1 + 10;
    Printf("Address of a = %u\n", p1);
    Printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);
    *p2 = *p2 + 3;
    *p1 = *p2 - 5;
```

```
printf("\n a = %d , b = %d ", a, b);
printf("z = %d \n", z);
}
```

o/p

Address of a = 4020 &

Address of b = 401b

a = 12 , b = 4

x = 42 , y = 9

a = 2 , b = 7 , z = 8 .

POINTER INCREMENTS AND SCALE FACTOR

The pointers can be incremented like :

```
P1 = P2 + 2 ;
P1 = P1 + 1 ;
```

and so on.

An expression like ,

```
P1++ ;
```

will cause the pointer P_1 to point to the next value of its type. For eg, if P_1 is an integer pointer with an initial value say 2800 , then after the operation $P_1 = P_1 + 1$, the value of P_1 will be 2802 and not 2801.

That is , when we increment a pointer , its value is increased by the length of the data type that it points to. This length is called scale factor.

The length of various data types are as follows:

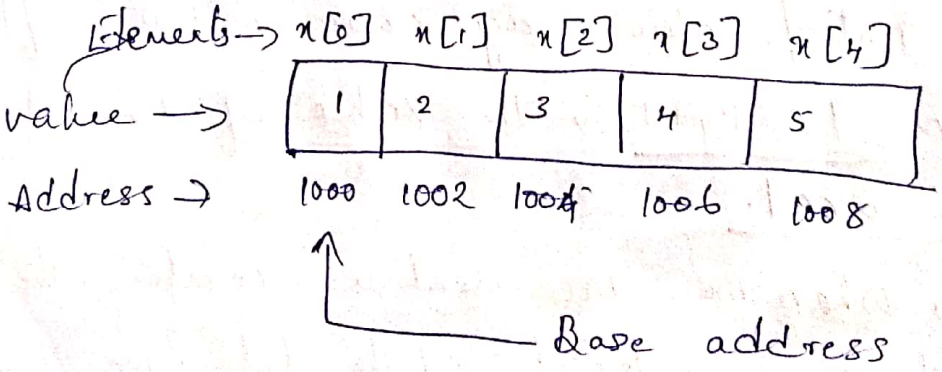
Characters	1 byte
Integers	2 bytes
floats	4 bytes
longintegers	4 bytes
doubles	8 bytes

POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contigous memory locations. The base address is the location of the 1st element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

Eg, `int x[5] = {1, 2, 3, 4, 5};`

Suppose the base address of x is 1000 and assuming that each integer requires 2 bytes, the 5 elements will be stored as follows:



name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored.

That is,

$$x = \&x[0] = 1000$$

If p is an integer pointer, we can make the pointer p to point to the array x by:

```
p = x ;
```

⟹ `p = &x[0];`

Similarly,

$$p = \&x[0] = 1000$$

$$p+1 = \&x[1] = 1002$$

$$p+2 = \&x[2] = 1004$$

Write a pgm using pointer to compute the sum of all elements stored in an array.

```
#include<stdio.h>
void main()
{   int *p, sum, i;
    int x[5] = {5,9,6,3,7};
    i = 0;
    p = x;
    printf("element value address\n");
    while(i<5){
    printf("x[%d] %d %u\n", i, *p, p);
    sum = sum + *p;
```

```
        i++;
        p++;
    }
    printf("\n sum = %d\n", sum);
    printf("\n &x[0] = %u\n", &x[0]);
    printf("\n p = %u\n", p);
}
```

⟹ Pointers can be used to manipulate 2 dimensional arrays as well.

`p` ⟶ pointer to first row.

`p+i` ⟶ pointer to i^{th} row.

`*(p+i)` ⟶ pointer to first element in the i^{th} row.

`*(p+i)+j` ⟶ pointer to j^{th} element in the i^{th} row.

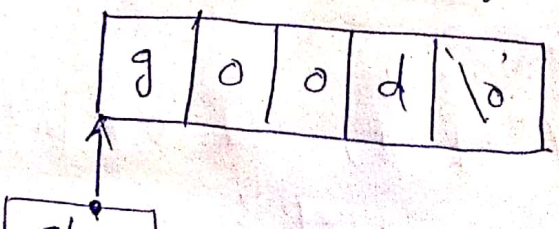
`*(*(p+i)+j)` ⟶ value stored in the cell (i, j)

POINTERS AND CHARACTER STRINGS

C supports an alternative method to create strings using pointer variables of type char. Example,

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable str. The pointer str now points to the first character of the string "good" as;



We can also use the run-time assignment for giving values to a string pointer. Example,

```
Char *string1;
string1 = "good";
```

We can print the content of the string string1 using either printf or puts functions as follows:

```
printf("%s", string1);
puts(string1);
```

A program to count the length of a string.

```
#include<stdio.h>
main()
{
    char *name;
    int length;
    char *cptr = name;
    name = "DELHI";
    printf("%s", name);
    while (*cptr != '\0')
    {
        printf("%c is stored at address %u \n",
                *cptr, cptr);
        cptr++;
    }
    length = cptr - name;
    printf("The length of the string is %d", length);
}
```

Output

DELHI

D is stored at address 54.
E is stored at address 55
L is stored at address 56
H is stored at address 57.
I is stored at address 58

ARRAYS OF POINTERS

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
Char name[3][25];
```

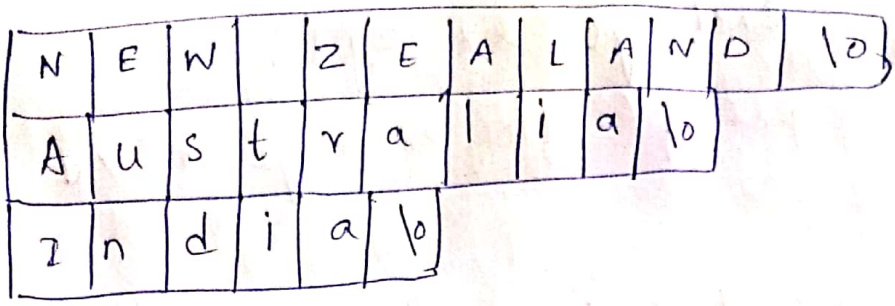
This says that name is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the Name table are 75 bytes.

For example

```
Char *name[3] = { "New Zealand", "Australia",
                  "India" };
```

declares name to be an array of 3 pointers to characters each pointer pointing to a particular name as:

name[0] ⟶ New Zealand
name[1] ⟶ Australia.
name[2] ⟶ India.



The following statement would print out all the three names;

```
for (i=0; i<=2; i++)
    printf ("%s\n", name[i]);
```

To access the j^{th} character in the i^{th} name, we may write as,

```
*(name[i]+j)
```

The character arrays with the rows of varying length are called ragged arrays and are better handled by pointers.

POINTERS AS FUNCTION ARGUMENTS.

A function using pointers to exchange the values stored in 2 locations in the memory.

Passing of pointers as function parameters

```
#include <stdio.h>
void exchange (int *, int *);
main()
{
    int x, y;
    x = 100;
    y = 200;
    printf ("Before exchange x=%d y=%d\n",
            x, y);
```

```
    exchange (&x, &y);
    printf ("After exchange x = %d  y = %d \n", x, y);
}

void exchange (int *a, int *b)
{ int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

output:

Before exchange : x = 100 y = 200

After exchange : x = 200 y = 100

The function sort may be written using pointers:

```
void sort (int m, int *x)
{
    int i, j, temp;
    for (i=1; i<=m-1; i++)
        for (j=1; j<=m-1; j++)
            if (*(x+j-1) >= *(x+j))
            {
                temp = *(x+j-1);
                *(x+j-1) = *(x+j);
                *(x+j) = temp;
            }
}
```

ng the call by reference method pgm to calculate the sum of 2 no:s which are passed as arguments.

pgm

```
#include <stdio.h>
#include <conio.h>
void swap (int *p, int *q);
void main()
{
    int x=10;
    int y=20;
    clrscr();
    printf("value of x & y before swapping x=%d &
                y=%d", x, y);
    swap(&x, &y);
    printf("\n value of x and y after swapping x=%d
            and y=%d", x, y);
}
void swap (int *p, int *q)
    {
        int t;
        t = *p;
        *p = *q;
        *q = t;
    }
```

output:

value of x and y before swapping are x=10 and y=20
value of x and y after swapping are x=20 and y=10

FUNCTIONS RETURNING POINTERS

Since pointers are a datatype in C, we can force a function to return a pointer to the calling function.

```
int *larger (int *, int *);
void main()
{
    int a = 10;
    int b = 20;
    int *p;
    p = larger(&a, &b);
    printf("%d", *p);
}
int *larger(int *x, int *y)
{
    if (*x > *y)
        return (x);
    else
        return (y);
}
```

POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as

follows :

type (*fptr)();

This tells the compiler that fptr is a pointer to a function, which returns type value. The parenthesis around *fptr is necessary. The statement like

type *gptr();

would declare gptr as a function returning a pointer to type.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. for example the statements

```
double mul(int, int);
double (*p1)();
p1 = mul;
```

declare P_1 as a pointer to a function and mul as a function and then make P_1 to point to the function mul. To call the function mul, we may now use the pointer P_1 with the list of parameters. That is

```
(*p1)(x, y)    /* function call */
```

is equivalent to

```
mul(x, y)
```

A Pgm that uses a function pointer as a fu
argument.

```
#include <math.h>
#define PI 3.1415926
double y(double);
double cos(double);
double table(double (*f)(), double, double, double);
void main()
{
    printf("Table of y(x) = 2*x*x - x+1\n\n");
    table(y, 0.0, 2.0, 5.0);
    printf("\n Table of cos(x) \n\n");
    table(cos, 0.0, PI, 0.5);
}
double table(double (*f)(), double min, double max,
                                        double step)
{   double a, value;
    for(a = min; a <= max; a += step)
    {
        value = (*f)(a);
        printf("%f  %f", a, value);
    }
}
double y(double x)
{
    return (2*x*x - x+1);
}
```

Output :

Table of $y(x) = 2*x*x - x + 1$

0.00	1.00000
0.50	1.0000
1.00	2.0000
1.50	4.0000
2.00	7.0000

Table of $\cos(x)$

0.00	1.0000
0.50	0.8776
1.00	0.5403
1.50	0.0707
2.00	-0.4161
2.50	-0.8011
3.00	-0.9900

Compatibility and casting

A pointer always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as incompatibility of pointers.

All the pointers variables store memory addresses which are compatible, but what is not compatible is the underlying data type to which they point to. We can however make explicit assignment between incompatible pointer types by using cast operator.

Example:

```
int x;
char *p;
p = (char *) &x;
```

We have an exception. The exception is the void pointer (void *). The void pointer is a generic pointer that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

POINTERS AND STRUCTURES

```
struct inventory
{
    char name[30];
    int number;
    float price;
} product[2], *ptr;
```

This statement declares product as an array of 2 elements, each of type struct inventory and ptr as a pointer to data objects of the type struct inventory. The assignment

```
ptr = product;
```

would assign the address of the zeroth element of product to ptr. That is, pointer ptr will now point to product[0]. Its members can be accessed using

```
ptr -> name;
ptr -> number;  ptr -> price
```

illustrate the use of structure pointers.

```
struct invent
{
    char *name[20];
    int number;
    float price;
};
main()
{
    struct invent product[3], *ptr;
    printf("INPUT \n");
    for (ptr = product; ptr < product+3; ptr++)
        scanf("%s %d %f", ptr->name, &ptr->number,
                                    &ptr->price);
    printf("\n OUTPUT \n");
    ptr = product;
    while (ptr < product+3)
    {
        printf("%s %d %f \n", ptr->name, ptr->
                number, ptr->price);
        ptr++;
    }
}
```

Output:

```
INPUT
WASHING_MACHINE    5    7500
TWO-IN-ONE         7    1250
OUTPUT
WASHING_MACHINE    5    7500
```

FILE MANAGEMENT IN C

The functions such as scanf and printf were used to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. The console oriented I/O operations pose two major problems.

1) It becomes cumbersome and time consuming to handle large volume of data through terminals
2) The entire data is lost when either the program is terminated or the computer is turned off.

A more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

A file is a place on the disk where a group of related data is stored. C supports a number of functions that have the ability to perform basic file operations, which include:

1) Naming a file.
2) Opening a file.
3) Reading data from a file.
4) Writing data to a file.
5) closing a file.

DEFINING AND OPENING A FILE

To store the data in the secondary memory, we must

They include the following:

1. Filename.
2. Data structure
3. purpose.

→ Filename is a string of characters that make up a valid file name for the operating system. It may contain 2 parts, a primary name and an optional period with the extension. Examples:

input.data
prog.c
Text.out

→ Data structure of the file is defined as FILE in the library of standard I/O function definitions. FILE is a defined data type. All files should be declared as type FILE before they are used.

→ The general format for declaring and opening a file:

```
FILE *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable fp as a "pointer to the data type FILE". The second statement opens the file named filename and assigns an identifier to the FILE type pointer *fp.

second statement also specifies the purpose of opening this file. Mode can be one of the following

r — open the file for reading only.

w — open the file for writing only.

a — open the file for appending (or adding) data to it.

When trying to open a file, one of the following things may happen:

1. When the mode is writing, a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is appending, the file is opened with the current contents safe. A file with the specified name is created if the ~~following~~ file does not exists.
3. If the purpose is reading, and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following example:

```
FILE *P1, *P2;
P1 = fopen ("data", "r");
P2 = fopen ("results", "w");
```

The file data is opened for reading and results is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If data file does not exists, an error will occur.

Many recent compilers include additional modes of operation. They include.

r+ — The existing file is opened to the beginning for both reading & writing.

w+ — Same as w except both for reading & writing.

a+ — Same as a except both for reading & writing.

CLOSING A FILE

A file must be closed as soon as all the operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all the links to the file are broken. It also prevents the accidental misuse of the file.

It takes the following form:

```
fclose (file_pointer);
```

```
FILE * p1;
p1 = fopen ("INPUT", "w");
- - -
fclose (p1);
```

INPUT / OUTPUT OPERATIONS ON FILES.

The getc and putc functions.

The simplest file I/O functions are getc and putc. They are analogous to getchar and putchar functions, and handle one character at a time. Assume that a file is opened with mode w and file pointer fp1. Then the statement

```
putc(c, fp1);
```

writes the character contained in the character variable c to the file associated with FILE pointer fp1.

Similarly, getc is used to read a character from a file that has been opened in read mode. For eg, the statement

```
c = getc(fp2);
```

would ~~write~~ read a character from the file whose file pointer is fp2. The getc will return an end-of-file marker EOF, when the end of file is reached.

WAP to read data from the keyboard, write it to a file called INPUT and again read the same & display it on the screen.

Pgm

```
#include<stdio.h>
main()
{ FILE *f1;
  Char c;
  Printf("Data input \n");
  f1 = fopen("INPUT", "w");
```

```
while ((c = getchar()) != EOF)
        putc(c, f1);
    fclose(f1)
printf("Data output \n");
    f1 = fopen("INPUT", "r");
        while ((c = getc(f1)) != EOF)
            printf("%c", c);
        fclose(f1);
}
```

The getw and putw functions

The getw and putw are integer-oriented functions. They are used to read & write integer values. The general forms are as follows:

```
putw(integer, fp);
getw(fp);
```

A program for demonstrating putw & getw.

```
#include<stdio.h>
main()
{   FILE *f1, *f2, *f3;
    int number, i;
    printf("contents of Data file \n");
    f1 = fopen("Data", "w");
```

```
for (i=1; i<=30; i++)
  {
      scanf ("%d", &number);
     if (number == -1) break;
      putw (number, f1);
  }
fclose (f1);
f1 = fopen ("Data", "r");
  f2 = fopen ("ODD", "w");
  f3 = fopen ("EVEN", "w");
while (number = getw(f1)) != EOF)
  { if (number % 2 == 0)
        putw (number, f3);
     else
        putw (number, f2);
  }
   fclose (f1);
   fclose (f2);
   fclose (f3);
  f2 = fopen ("ODD", "r");
   f3 = fopen ("EVEN", "r");
   printf ("\n contents of ODD file \n");
      while (number = getw (f2)) != EOF)
        printf ("%d", number);
```

```
printf ("\n contents of Even file \n");
    while ((number = getw(f3)) != EOF)
        printf ("%d", number);
    fclose (f2);
    fclose (f3);
}
```

o/p

Contents of DATA file.

111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

CONTENTS OF ODD FILE -

111 333 555 777 999 121 343 565

CONTENTS OF EVEN FILE.

222 444 666 888 0 232 454

The fprintf and fscanf functions

fprintf and fscanf can handle a group of mixed data simultaneously. The general form of fprintf is.

fprintf (fp, "control string", list);

where fp is a file pointer associated with a file that has been opened for writing. The control string contains output specifications for the items in the list. The list may include variables, constants and strings.

Eg: fprintf (f1, "%s %d %f", name, age, 7.5);

here name is an array variable of type char and age is an int variable.

The general format of fscanf is

fscanf(fp, "control string", list);

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.

Eg:

fscanf(f2, "%s %d", item, &quantity);

When the end of file is reached, it returns EOF.

Write a pgm to open a file named INVENTORY and store in it the following data.

item name	number	price	quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

→ The filename INVENTORY is supplied through the keyboard.

→ Data is read using the function fscanf from the file stdin, which refers to the terminal and then it is written to the file that is being pointed to by the filepointer fp.

After closing the file INVENTORY, it is again reopened for reading. The data from the file along with the item values are written to the file stdout, which refers to the screen. While reading from a file, it is read to the end of file.

Pgm

```
#include<stdio.h>
main()
{
    FILE *fp;
    int number, quantity, i;
    float price, value;
    char item[6], filename[10];

    printf("input file name \n");
    scanf("%s", filename);
    fp = fopen(filename, "w");
    printf("input inventory data \n");
    printf("item - name number price quantity \n");
    for (i=1; i<=3; i++)
        {
            fscanf(stdin, "%s%d%f%d", item, &number,
                                &price, &quantity);
            fprintf(fp, "%s%d%f%d", item, number, price,
                                quantity);
        }
    fclose(fp);
```

```
fprintf (stdout , "\n\n");
fp = fopen (filename, "r");
printf(" item name number price quantity value\n");
for (i=1 ; i<=3 ; i++)
{
    fscanf (fp, "%s %d %f %d", item , & number , & price ,
                                & quantity);
    value = price * quantity;
    fprintf (stdout , "%s %d %f %d %f", item, number ,
                price , quantity , value);
}
fclose (fp);
}
```

ERROR HANDLING DURING I/O OPERATIONS

Typical error situations include the following:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid name.
6. Attempting to write to a write protected file.

feof and ferror → 2 status-inquiry library functions that can help us detect I/O errors in the files.

→ The feof function can be used to test for an end of file condition. It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If fp is a pointer to file that has just been opened for reading, then the stmt,

```
if (feof(fp))
    printf("End of data\n");
```

would display the message "End of data" on reaching the end of file condition.

→ The ferror function reports the status of the file indicated. It also takes a FILE pointer as its argument and returns a non-zero integer if an error has been detected upto that point, during processing. It returns zero otherwise

```
if (ferror(fp) != 0)
    printf("An error has occured\n");
```

→ Whenever a file is opened for reading/writing/appending, a file pointer is returned. If the file cannot be opened it returns a NULL pointer.

```
if (fp == NULL)
    printf("File could not be opened\n");
```

write a pgm to illustrate error handling in file operations

pgm

```
#include<stdio.h>
main()
{
    char *filename;
    FILE *fp1, *fp2;
    int i, number;
    fp1 = fopen("Test", "w");
    for(i=10; i<=100; i+=10)
        putw(i, fp1);
    fclose(fp1);
    printf("\n input filename\n");
    open_file:
        scanf("%s", filename);
    if((fp2 = fopen(filename, "r"))==NULL)
    {
        printf("cannot open the file\n");
        printf("Type filename again\n");
        goto open_file;
    }
    else
        for(i=1; i<=20; i++)
        {
            number = getw(fp2);
```

```
if (feof (fp2))
  { printf ("\n Ran out of data\n");
    break;
  }
else
  printf ("%d\n", number);
}
fclose(fp2);
}
```

RANDOM ACCESS TO FILES

→ Only a particular part of a file and not reading the other parts.

→ These can be done by functions like fseek, ftell and rewind.

→ ftell takes a file pointer and returns a number of type long, that corresponds to the current position. This function is useful in saving the current position of a file.

It takes the following form:

```
n = ftell (fp);
```

n would give the relative offset (in bytes) of the current position. This means that n bytes have already been read or written.

→ rewind takes a file pointer and resets the position to the start of the file.

For eg: , the statement

```
rewind (fp);
n = ftell (fp);
```

would assign 0 to n because the file position has been set to the start of the file by rewind. Whenever a file is opened for reading or writing, a rewind is done implicitly.

→ fseek function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek (file_ptr, offset, position);
```

→ file_ptr is a pointer to the file concerned.
→ offset is a number or variable of type long.
→ position is an integer number.

offset specifies the number of position (bytes) to be moved from the location specified by position.

The position can take one of the following 3 values:

value	meaning
0	Beginning of file
1	Current position.
2	End of file.

When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and fseek returns -1.

Operations of fseek function

stmt	meaning
1. fseek(fp, 0L, 0);	Go to the beginning.
2. fseek(fp 0L, 1);	Stay at current position.
3. fseek(fp, 0L, 2);	Go to the end of the file, past the last character of the file.
4. fseek(fp, m, 0);	Move to (m+1)th byte in the file.
5. fseek(fp, m, 1);	Go forward by m bytes.
6. fseek(fp, -m, 1);	Go back ~~back~~ward by m bytes from the current position.
7. fseek(fp, -m, 2);	Go backward by m bytes from the end.

A sample program

```
#include <stdio.h>
main()
{ FILE *fp;
    long n;
    char c;
    fp = fopen("RANDOM", "w");
    while ((c = getchar()) != EOF)
        putc(c, fp);
    printf("No: of characters entered = %d\n", ftell(fp));
    fclose(fp);
```

```
fp = fopen("RANDOM", "r");
n = 0L;
while (feof(fp) == 0)
{
  fseek(fp, n, 0);
  printf("position of %c is %d\n", getc(fp), ftell(fp));
  n = n + 5L;
}
putchar('\n');
fseek(fp, -1L, 2);
do
{
  putchar(getc(fp));
} while(!fseek(fp, -2L, 1));
fclose(fp);
}
```

output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ ^Z
No: of characters entered = 26
position of A is 0.
position of F is 5.
position of K is 10.
position of P is 15.
position of U is 20.
position of Z is 25
Poston of  is 8
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

COMMAND LINE ARGUMENTS

→ What is a command line argument?

→ It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process.

→ For example, if we want to execute a program to copy the contents of a file named x-file to another one named y-file, then we use a command line like

C > PROGRAM x-file y-file

where program is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution.

→ Every C program should have one main function and that marks the beginning of the program. Main can take 2 arguments called argc and argv and the information contained in the command line is passed on to the program through these arguments, when main is called up by the system.

→ The variable argc is an argument counter that counts the number of arguments on the command line.

→ The argv is an argument vector and represents an array of character pointers that point to the command line arguments.

The size of this array will be equal to the value of argc. For example, for the command line given below,

⟶ C > PROGRAM X_FILE Y_FILE

argc is three

argv is an argument vector and represents an array of character pointers that point to the command line arguments. So argv is an array of 3 pointers to strings like given below.

argv[0] → PROGRAM.

argv[1] → X_FILE.

argv[2] → Y_FILE.

Inorder to access the command line arguments, we must declare main like this:

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

The first parameter in the command line, is always the program name and argv[0] always represents the program name.

Write a pgm that will receive a filename and a line of text as command line arguments and write the first to the file.

```
#include <stdio.h>
main (int argc, char *argv[])
{
  FILE *fp;
  int i;
  char word[15];
  fp = fopen (argv[1], "w");
  printf ("No: of arguments in command line = %d", argc);
  for (i=2; i<argc; i++)
    fprintf (fp, "%s", argv[i]);
  fclose (fp);
  printf ("Contents of the %s file \n", argv[1]);
  fp = fopen (argv[1], "r");
  for (i=2; i<argc; i++)
    {
      fscanf (fp, "%s", word);
      printf ("%s", word);
    }
  fclose (fp);
}
```

C > FIL7 TEXT AAAA BBBB CCCC DDDD EEEE FFFF GGGG

FUNCTIONS

C functions can be classified into 2 categories. — Library functions
— user-defined functions.

main is an eg. of user defined functions. printf & scanf belong to the category of library functions.
Every program must have a function to indicate where the pgm has to begin its execution.

Elements of User defined functions:

1. → Function Definition
2. → Function Call
3. → Function declaration.

Function definition is an independant program module that is specially written to implement the rqmnts of the function.

Function call: In order to use the function we need to invoke it at a required place in the programs.

Function declaration: The calling program should declare any function that is to be used later in the pgm. This is known as the function declaration or function prototype.

Definition of Functions:

A general format of a function definition to implement 2 parts:

```
function-type function_name (parameter list)
{ local variable declaration;
  executable stmt;
  executable stmt;
  return statement;
}
```

The first line function-type function_name (parameter list)

is known as the function header and the stmts within the opening & closing braces constitute the function body, which is compound statement.

function-type: specifies the type of value that the function is expected to return to the program calling the function. If the return type is not explicitly specified C will assume that is an integer type. If the function is not returning anything, then specify the return type as void.

Function name is any valid C identifier.

Return values & Their Types:

The return stmt can take one of the following forms: return; or return(expression);

→ The plain return doesn't return any value. it acts as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

```
if(error)
    return;
```

→ The second form of return with an expr. returns the value of the expr. For eg:

```
int mul(int x, int y)
{ int p;
  p= x*y;
} return(p);
```

→ A function may have more than one return stmts.

```
if(x<=0)
    return(0);
else
    return(1);
```

Function Calls:

A function can be called by simply using the function name followed by a list of actual parameters (arguments), if any, enclosed in parentheses.

Eg:
```
main()
{ int y;
  y= mul(10, 5);
  printf("%d \n", y);
}
main()
{
  int y;
  y= mul(10, 5); /* call */
}
  int mul(int x, int y)
  { int p;
    p= x*y;
    return(P);
  }
```

Function Declaration:

Format: function-type fun-name(parameter-list);

Eg: int mul(int m, int n); /* function prototype */

Points to remember:

→ The parameter list must be separated by commas.

→ The parameter names do not need to be the same in the prototype declaration & the function definition.

→ The types must match the types of parameters in the function definition, in number & order.

→ Use of parameter names in the declaration is optional.

→ If the function has no formal parameters, the list

→ The return type is optional, when the function returns int type data.
→ The return type must be void if no value is returned.
→ When the declared types do not match with the types in fn definition, compiler will produce an error.

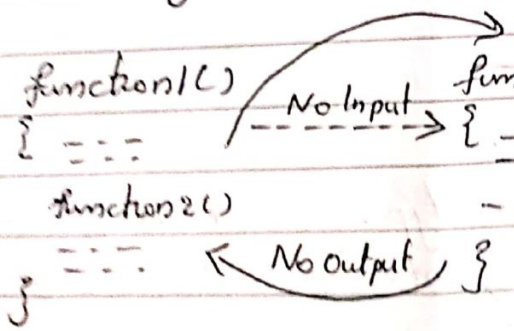
Equally acceptable forms of declaration of mul fn. are

```
int mul(int, int);
mul(int a, int b);
mul(int, int);
void display(void);
```

A prototype declaration may be placed in 2 places in a program
1. Above all the functions (including main)
2. Inside a function definition.

Category of Functions:

① → No Arguments & No Return Values:



// No data comm^n b/w fns.

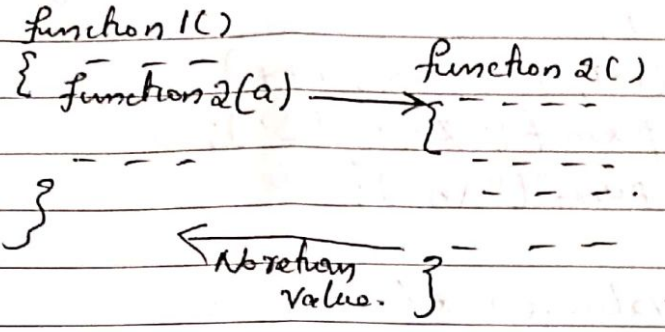
```
/* function declaration */
void printline(void);
void value(void);
main()
{ printline();
  value();
  printline();
}
/* printline() */
```

```
void printline(void) /* contains no arguments */
{ int i;
  for(i=1; i<=35; i++)
    printf("%c", '_');
    printf("\n");
}
/* value() */
void value(void)
{ int yr, period;
  float inrate, sum, principal;
  printf("Principal Amount");
  scanf("%f", &principal);
  printf("Interest rate?");
  scanf("%f", &inrate);
  printf("period");
  scanf("%d", &period);
  sum=principal;
  yr=1;
  while(yr<=period)
  { sum = sum*(1+inrate);
    yr=yr+1;
  }
  printf("\n%f %f %d %f\n", principal
          inrate, period, sum);
}
```

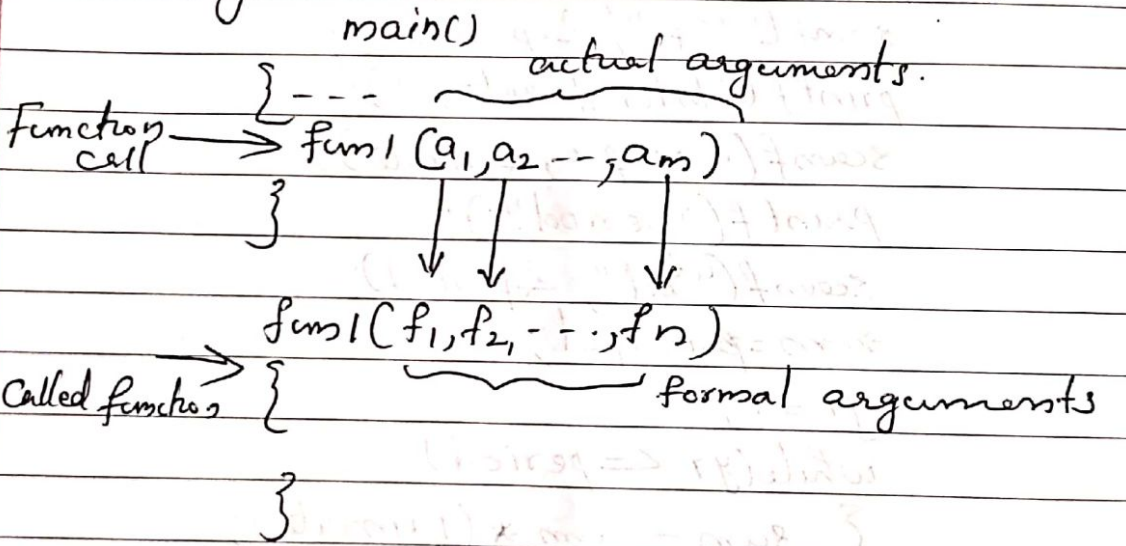
Output Principal

```
principal amount? 5000
Interest rate?    0.12
Period?           5
5000.00 0.12 5 8811.71
```

② Arguments But no Return Values:

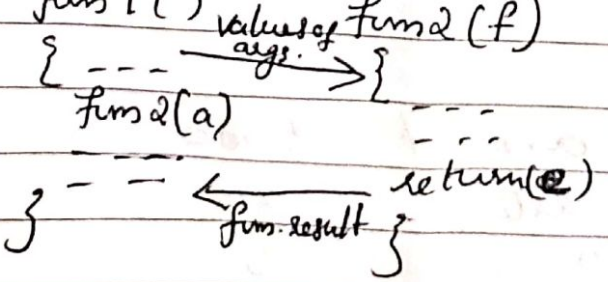


Arguments in function call is called actual arguments Arguments in function definition is called formal arguments.



Eg: void printline (char c);
void value (float, float, int);
main()
{
float

③ Arguments with Return Values:



④ No Arguments But Returns Value.

```
Eg: int get_number(void);
    main()
    { int m=get_number();
      printf("%d", m);
    }
    int get_number(void)
    { int number;
      scanf("%d", &number);
      return(number);
    }
```

5/9/19

Functions that return multiple Values:

— return one value using a return stmt.

A return stmt can return only one value.

The mechanism of sending back info through args. is achieved using what are known as the address operator (&) & indirection operator (*).

```
void mathoperation(int x, int y, int *s, int *d);
main()
{ int x=20, y=10, s, d;
  mathoperation(x, y, &s, &d);
  printf("s=%d\n d=%d\n", s, d);
}
void mathoperation(int a, int b, int *sum, int *diff)
{ *sum = a+b;
  *diff = a-b;
}
```

When the fn is called the follo. assignments occur:

value of x to a

value of y to b

address of s to sum address of d to diff.

RECURSION

Recursion is a process where a function calls itself.

Eg:
```
main()
{ printf("Recursion");
  main();
}
```

O/P: Recursion Recursion ....

Execution is terminated abruptly.

Factorial: i.e; eg: 5!

$= 5 \times 4 \times 3 \times 2 \times 1$

$= 5 \times 4!$

$= 5 \times 4 \times 3!$

$= 5 \times 4 \times 3 \times 2!$

$= 5 \times 4 \times 3 \times 2 \times 1!$

$= 5 \times 4 \times 3 \times 2 \times 1$

Eg:
```
factorial(int n)
{ int fact;
  if(n==1)
     return(1);
  else
     fact = n*factorial(n-1);
  return fact;
}
```

n = 3.

fact = 3 * factorial(2);

fact = 3 * 2 * factorial(1);

fact = 3 x 2 x 1;

= 6

PASSING ARRAYS to FUNCTIONS

One Dimensional Arrays:

To pass a one-dimensional array to a called function, it is sufficient to list the name of the array, without any subscripts, & the size of the array as arguments. For eg:

the call largest(a, n) will pass the whole array a to the called function. The called fn. expecting this call must be appropriately defined. The largest fn header might be like this.

float largest(float array[], int size)

Eg: Largest of array elements:

```
main()
{ float largest(float a[], int n);
  float value[4] = {2.5, -4.75, 1.2, 3.67};
  printf("%f\n", largest(value, 4));
}
  float largest(float a[], int n)
    {
      int i
      float max;
      max = a[0];
      for(i=1; i<n; i++)
          if(max < a[i])
          max = a[i];
          return(max);
      }
```

Write a program that uses a function to sort an array of integers.

Program

```
void sort(int m, int x[]);
main()
{ int i;
  int marks[5] = {40, 90, 73, 81, 35};
  printf("Marks be~
  sort(5, marks);
  printf("\n After sorting");
  for(i=0; i<5; i++)
      printf("\n%d", marks[i]);
}
void sort(int m, int x[])
{ int i, j, t;
  for(i=1; i<=m-1; i++)
    for(j=1; j<=m-i; j++)
      if(x[j-1] >= x[j])
        {
          t = x[j-1];
          x[j-1] = x[j];
          x[j] = t;
        }
}
```

Output:

```
Marks After sorting
35  40  73  81  90.
```

Two dimensional Arrays:

Rules: → The function must be called by passing only the array name.

→ In the fn definition, we must indicate that the array has 2 dimensions by including 2 sets of brackets.

→ The size of the 2nd dimension must be specified.
→ The prototype declaration should be similar to the fn. header.

```
Eg: double average (int x[][N], int M, int N)
    { int i,j;
      double sum = 0.0;
      for (i=0; i<M; i++)
        for (j=1; j<N; j++)
          sum += x[i][j];
          return (sum/(M*N));
    }
```

Passing strings to Functions:

The string to be passed must be declared as a formal arg. of the fn. when it is defined:

```
Eg: void display (char item_name[])
    { ...
    }
```

The fn prototype must show that the arg. is a string

```
void display (char str[]);
```

A call to the fn must have a string array name without subscripts as its actual args.

```
display (names);
```

Parameter Passing Methods

The technique used to pass data from one fn to another is known as parameter passing. It can be done in 2 ways:
* Pass by value (call by value)
* Pass by reference/pointers.

Pass by value: values of actual parameters are copied to the variables in the parameter list of the called fn. The called fn works on the copy & not on the original values

of the actual parameters. This ensures that the original data in the calling fn cannot be changed accidentally.

In pass by pointers (pass by address/ref.), the memory addresses of the variables rather than copies of values are sent to the called fn. In this case, the called fn. directly works on the data in the calling fn & the changed values are available in the calling fn for its use.

Pass by pointers method is used when manipulating arrays & strings. This method is also used when we require multiple values to be returned by the called function.

STORAGE CLASSES

The following variable storage classes are most relevant to functions:

1. Automatic variables
2. External variables
3. Static variables
4. Register variables.

The scope of variable determines over what region of the pgm a variable is actually available for use. Longevity refers to the period during which a variable retains a given value during exn of a pgm.
The visibility refers to the accessibility of a variable from the memory.

Automatic Variables:

They are declared inside a when the function in which they are to be utilized. They are created when the fn is called and destroyed automatically when the fn is exited, hence the name is automatic.
← local or internal variables.

A variable declared inside a fn without storage class specification is, by default, an automatic variable.

```
main()
{ int number;
  ---
}                    // number is auto variable.

main()
{ int auto int number;
  ---
}                    // explicitly declared.
```

Eg:

```
void fun1(void);
void fun2(void);
main()
{ int m=1000;
  fun2();
  printf("%d\n", m); /* Third o/p */
}
  void fun1(void)
  { int m=10;
    printf("%d\n", m); /* First o/p */
  }
  void fun2()
  { int m=100;
    fun1();
    printf("%d\n", m); /* Second o/p */
  }
```

O/p:

```
10
100
1000
```

Their value cannot be changed accidentally by what happens in some other fn in the pgm. This assures that we may declare & use same variable

name in diff. fns in the same pgm without causing any confusion to the compiler.

External Variables:

Variables that are both alive & active throughout the entire program are known as external variables. They are global variables.

Global variables can be accessed by any fn in the programs. External variables are declared outside a function.

```
int number;
float length = 7.5;
main()
{ - - - -
  - - - -
}
fun1()
{ - - -
  - - -
}
  fun2()
  { - - -
  } - - -
```

The variables number & length are available for use in all the 3 functions. In case, a local variable & a global variable have the same name, the local variable will have precedence over the global one in the fn where it is cleared.

```
int count;
main()
{ count = 10;
  - - -
}
fun()
 { int count = 0;
   - - -
   count = count + 1;
}
```

When fun1() references the variable count, it will be referencing only its local variable, not the global one. The value of count in main will not be affected.

```
int fun1(void);
int fun2(void);
int fun3(void);
int x; /* global */
main()
{ x=10; /* global */
  printf("x=%d", x);
  printf("x=%d", fun1());
  printf("x=%d", fun2());
  printf("x=%d", fun3());
}
fun1(void)
 { x=x+10;
 }
fun2(void)
 { int x; /* local */
   x=1;
   return x;
 }
fun3(void)
 { x=x+10; /* global x */
 }
```

O/P
x=10
x=20
x=1
x=30

External Declaration.

```
main()
{ y=5;
  ---
}
---
int y; /* global declaration */
fun1()
{ y=y+1;
}
```

As far as main is concerned, y is not defined. So the compiler will issue an error msg. Unlike local variables, global variables are initialized to zero by default. The stmt y=y+1; will assign 1 to y.

```
main()
{ extern int y; /* external declaration */
  ---
}
---
fun1()
{ extern int y; /* external declaration */
  ---
}
---
int y; /* definition */
```

The external declaration of y inside the functions informs the compiler that y is an integer type defined somewhere else in the pgm. Note that extern declaration doesn't allocate storage space for variables.

An extern within a function provides the type information to just that one function. We can provide type inf^n to all functions within a file by placing external declarations before any of them.

```
extern float height[];
main()
{ int i;
```

```
void print(void);
  - - -
  - - -
print();
}
void print()
{ int i;
  - - -
}
float height[SIZE];
```

The distinction b/w definition and declaration also applies to functions. A function is defined when its parameters & function body are specified. This tells the compiler to allocate space for the function code & provides type info for the parameters. Since funcs functions are external by default, we declare them without the qualifier extern. ∴ the declaration

void print(void); is equivalent to

extern void print(void);

Static Variables:

The value of static variables persists until the end of the programs. A variable can be declared static using the keyword static like

```
static int x;
static float y;
```

Eg:

```
void stat(void);
main()
{ int i;
for(i=1; i<=3; i++)
stat();
}
```

```
void stat(void)
{
static int x=0;
x=x+1;
printf("x=%d", x);
}
```

O/P:

x=1
x=2
x=3.

declared as an auto variable, the o/p would have been

x=1, x=1, x=1

Register Variables

→ A variable should be kept in one of the mc's registers instead of keeping in the memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables (loop control variables) in the register will lead to faster exn of pgms.

register int count;

Most compilers allow only int or char variables to be placed in the registers.

Library Functions:

C functions can be classified into 2 categories, Library functions & user-defined functions. main is an eg: of user-defined functions. printf & scanf belong to the category of library functions.

eg: sqrt, strcat etc.

The main distinction b/w these 2 categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a programs.

CONTENT BEYOND SYLLABUS

Dynamic memory allocation

C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

C malloc() method

"malloc" or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

C calloc() method

"calloc" or **"contiguous allocation"** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

Syntax:

ptr = (cast-type*)calloc(n, element-size);

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

C free() method

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);

C realloc() method

"realloc" or **"re-allocation"** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.