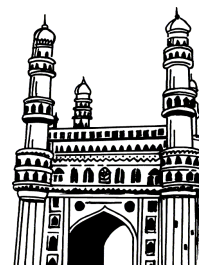


Rahul's ✓
Topper's Voice

AS PER
CBCS SYLLABUS



B.Sc.

I Year I Sem

(Osmania University)

Latest 2021 Edition

PROGRAMMING IN C

- ☞ Study Manual
- ☞ FAQ's and Important Questions
- ☞ Short Question & Answers
- ☞ Multiple Choice Questions
- ☞ Fill in the blanks
- ☞ C Lab Programs
- ☞ Solved Previous Question Papers
- ☞ Solved Model Papers

- by -

WELL EXPERIENCED LECTURER



Rahul Publications™

Hyderabad. Ph : 66550071, 9391018098

All disputes are subjects to Hyderabad Jurisdiction only

B.Sc.

I Year I Sem

(Osmania University)

PROGRAMMING IN C

In spite of many efforts taken to present this book without errors, some errors might have crept in. Therefore we do not take any legal responsibility for such errors and omissions. However, if they are brought to our notice, they will be corrected in the next edition.

© No part of this publication should be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise without the prior written permission of the publisher

Price ` 189-00

Sole Distributors :

☎ : 66550071, Cell : 9391018098

VASU BOOK CENTRE

Shop No. 3, Beside Gokul Chat, Koti, Hyderabad.

Maternity Hospital Opp. Lane, Narayan Naik Complex, Koti, Hyderabad.

Near Andhra Bank, Subway, Sultan Bazar, Koti, Hyderabad -195.

PROGRAMMING IN C

STUDY MANUAL

FAQ's and Important Questions	V - VIII
Unit - I	1 - 103
Unit - II	104 - 162
Unit - III	163 - 220
Unit - IV	221 - 256
C Programs	257 - 272

SOLVED PREVIOUS QUESTION PAPERS

October / November - 2020	273 - 275
October / November - 2019	276 - 279
May / June - 2019	280 - 281
November / December - 2018	282 - 283
May / June - 2018	284 - 285
November / December - 2017	286 - 287
May / June - 2017	288 - 289
December - 2016	290 - 290

SOLVED MODEL PAPERS

MODEL PAPER - I	291 - 292
MODEL PAPER - II	293 - 294
MODEL PAPER - III	295 - 296

SYLLABUS

UNIT - I

Computer Fundamentals: Introduction of Computers, Classification of Computers, Anatomy of a Computer, Memory Hierarchy, Introduction to OS, Operational Overview of a CPU. Program fundamentals: Generation and Classification of Programming Languages, Compiling, Interpreting, Loading, Linking of a Program, Developing Program, Software Development. Algorithms: Definitions, Different Ways of Stating Algorithms (Step-form, Pseudo-code,

Flowchart), Strategy for Designing Algorithms, Structured Programming Concept.

Basics of C: Overview of C, Developing Programs in C, Parts of Simple C Program, Structure of a C Program, Comments, Program Statements, C Tokens, Keywords, Identifiers, Data Types, Variables, Constants, Operators and Expressions, Expression Evaluation–precedence and associativity, Type Conversions.

UNIT - II

Input-Output: Non-formatted and Formatted Input and Output Functions, Escape Sequences, Control Statements: Selection Statements – if, if-else, nested if, nested if-else, comma operator, conditional operator, switch; Iterative Statements–while, for, do-while; Special Control Statement–goto, break, continue, return, exit. Arrays and Strings: One-dimensional Arrays, Character arrays, Functions from ctype.h, string.h, Multidimensional Arrays.

UNIT - III

Functions: Concept of Function, Using Functions, Call-by-Value Vs Call-by-reference, Passing Arrays to Functions, Scope of Variables, Storage Classes, Inline Functions, and Recursion.

Pointers in C: Introduction, Address of Operator (&), Arrays and Pointers, Pointers and Strings, Pointers to Pointers, Array of Pointers, Pointer to Array, Dynamic Memory Allocation.

UNIT - IV

User-defined Data Types: Declaring a Structure (Union) and its members, Initialization Structure (Union), Accessing members of a Structure (Union), Array of Structures (Union), Structures Vs Unions, Enumeration Types. Files in C: Introduction, Using Files in C, Working with Text Files, Working with Binary Files, Files of Records, Random Access to Files of Records, Other File Management Functions.

Contents

Topic	Page No.
UNIT - I	
1.1 Introduction to Computers	1
1.2 Classification of Computers	9
1.3 Anatomy of a Computer	13
1.4 Memory Hierarchy	16
1.5 Operating Systems	21
1.5.1 Introduction to Operating Systems	21
1.5.2 Types of Operating System	23
1.6 Operational Overview of a CPU	25
1.7 Program Fundamentals	27
1.7.1 Generation of Programming Languages	27
1.7.2 Generations / Types of Programming Languages	29
1.8 Compiling, Interpreting, Loading, Linking of a Program	32
1.9 Developing a Program	35
1.10 Software Development	37
1.11 Algorithms	39
1.11.1 Definition	39
1.12 Different Ways of Stating Algorithms	40
1.13 Strategy for Designing Algorithms	47
1.14 Structured Programming Concept	48
1.14.1 Top-Down Analysis	49
1.14.2 Modular Programming	50
1.14.3 Programming Process	51
1.15 Overview of C	53
1.16 Developing Programs in C	54
1.17 Parts of Simple C Program	58
1.18 Structure of a C Program	60
1.19 Comments	65
1.20 Program Statements	65
1.21 C Tokens	67
1.22 Keywords and Identifiers	68
1.23 Data Types	69

Topic	Page No.
1.24 Variables	72
1.25 Constants	74
1.25.1 Symbolic Constants	76
1.26 Operators	77
1.27 Expression Evaluation	86
1.28 Precedence and Associativity	86
1.29 Type Conversions	88
➤ Short Question and Answers	91 - 99
➤ Choose the Correct Answers	100 - 102
➤ Fill in the blanks	103 - 103

UNIT - II

2.1 Input and Output Operations	104
2.1.1 Non-formatted / formatted Input/Output Operations	105
2.2 Escape Sequences	112
2.3 Control Statements	112
2.3.1 Selection Statements / Decision Making Statements / Branching Statements	113
2.3.1.1 <i>if</i> control statement	114
2.3.1.2 <i>if... else</i> Control Statement	115
2.3.1.3 Nested <i>if...else, if...else if</i> Statements	116
2.3.1.4 Switch Statement	118
2.3.1.5 Comma Operator	119
2.3.1.6 Conditional Operator	119
2.4 Looping Statements	120
2.4.1 while loop	120
2.4.2 do-while loop	121
2.4.3 for Loop	123
2.5 Special Control Statements (<i>goto, break, continue, return, exist</i>)	125
2.6 Arrays & Strings	131
2.6.1 One Dimensional Arrays	131
2.6.2 Initialization of One Dimensional Array	134
2.6.3 Two/Multidimensional Arrays	136
2.7 Character Array	141
2.7.1 Declaring and Initializing Strings	142

Topic	Page No.
2.7.2 Reading Strings from the Terminal	144
2.7.3 Arithmetic Operations on Characters	146
2.8 FUNCTIONS from ctype.h, string.h	147
2.8.1 Array of Strings	150
➤ Short Question & Answers	152 - 159
➤ Choose the Correct Answers	160 - 161
➤ Fill in the blanks	162 - 162

UNIT - III

3.1 Functions	163
3.1.1 Concept of Functions	163
3.1.2 Elements of Functions	164
3.1.3 Defining Functions	165
3.1.4 Return Values and Their Types	166
3.1.5 Function Calls	167
3.1.6 No Arguments and No Return Values	169
3.1.7 Arguments with No Return Values	171
3.1.8 Arguments with Return Values	172
3.2 Using Functions	174
3.3 Call-by-Value Vs Call-by-Reference	176
3.3.1 Call by Value Vs Call by Reference	179
3.4 Passing Arrays to Functions	180
3.5 Scope of Variables	181
3.6 Storage Classes	185
3.6.1 Storage Class Specifiers	187
3.7 Inline Function	190
3.8 Recursion Functions	191
3.9 Pointers Introduction	194
3.9.1 Declaring Pointer Variables	198
3.9.2 Initialization of Pointer Variables	198
3.10 Address of operator	202
3.11 Uses of Pointers	203
3.12 Pointers and Arrays	203
3.13 IPointers and Character Strings	205

Topic Page No.

3.14	Dynamic Memory Allocation	208
➤	Short Question and Answers	212 - 217
➤	Choose the Correct the Answers	218 - 219
➤	Fill in the Blanks	220 - 220

UNIT - IV

4.1	User-Defined Data Types	221
4.1.1	Structures	221
4.1.2	Declaring Structure and its Members	222
4.2	Initialization of structure	225
4.3	Accessing Members of a Structure	226
4.4	Unions	232
4.5	Structures Vs Union	233
4.6	Enumeration Types	234
4.7	Files in C	236
4.8	Using files in C	240
4.9	Working with Text Files	242
4.10	Working With Binary Files	244
4.11	Random access to Files of Records	244
4.12	Other File Management Functions	246
➤	Short Question & Answers	249 - 253
➤	Choose the Correct Answers	254 - 255
➤	Fill in the blanks	256

Frequently Asked & Important Questions

UNIT - I

1. Define operating system ? Explain list the components of OS.

Ans : (June-17)

Refer Unit-I, Q.No. 15

2. What is a programming language? Explain their classifications?

Ans : (Nov.-20, June-19, Imp.)

Refer Unit-I, Q.No. 18

3. Explain different types of programming languages ?

Ans : (Nov.-20, June-19, June-18, Imp.)

Refer Unit-I, Q.No. 19

4. What is an algorithm? Explain the characteristics & steps for writing algorithm ?

Ans : (Dec.-18, Dec.-17)

Refer Unit-I, Q.No. 24

5. What are flowcharts ? Explain its types? What are the Symbols used in ? flowcharts

Ans : (June-18, Dec.-18, June-17, Imp.)

Refer Unit-I, Q.No. 26

6. Explain the structure of C Program?

Ans : (Nov.-20, June-19, Dec.-18, Dec.-17, June-17, Imp.)

Refer Unit-I, Q.No. 39

7. What do you mean by datatype ? How many data types does C - Language consist ?

Ans : (Dec.-19, June-18, Dec.-17)

Refer Unit-I, Q.No. 46

8. What are constants ? Explain its types?

Ans : (Dec.-19, Dec.-17)

Refer Unit-I, Q.No. 48

9. Define operator in C ? Explain all types of operators available in C with example?

Ans : (Dec.-18, June-18, Dec.-16, Imp.)

Refer Unit-I, Q.No. 50

10. Explain the precedence and Associativity of operators?

Ans : (June-19, June-18, Imp.)

Refer Unit-I, Q.No. 52

UNIT - II

1. Explain non-formatted & formatted I/O operations ?

Ans : (June-19, Dec.-18, June-18, Imp.)

Refer Unit-II, Q.No. 2

2. Explain about Selection Statements.

Ans : (Dec.-18)

Refer Unit-II, Q.No. 5

3. Explain Special statement in loops of C - language.

Ans : (Nov.-20)

Refer Unit-II, Q.No. 17

4. Define arrays ? What is the need of arrays ?

Ans : (June-19, Dec.-18)

Refer Unit-II, Q.No. 18

5. Explain the declaration and Accessing of one dimensional array ?

Ans : (June-19, Dec.-18, Imp.)

Refer Unit-II, Q.No. 19

6. What is Multidimensional Arrays ? Explain the intialization and accessing of Multidimensional Arrays.

Ans : (Nov.-20, Dec.-19, June-19, Dec.-18, Dec.-17, Imp.)

Refer Unit-II, Q.No. 21

7. What are strings ? Explain with examples ?

Ans : (June-19, Dec.-18, Imp.)

Refer Unit-II, Q.No. 23

8. Explain the process of declaring and initializing strings.

Ans : (June-19)

Refer Unit-II, Q.No. 24

9. Write the builtin function in string.h header file.

Ans : (Dec.-18, Dec.-17, Dec.-16, Imp.)

Refer Unit-II, Q.No. 27

UNIT - III

1. Explain call by value and call by reference with example ?

Ans : (Dec.-17, June-17, Imp.)

Refer Unit-III, Q.No. 11

2. Difference between call by value & call by reference ?

Ans : (June-19)

Refer Unit-III, Q.No. 12

3. Explain the method of passing arrays to functions ?

Ans : (Dec.-18, Dec.-17, Imp.)

Refer Unit-III, Q.No. 13

4. Explain the use of different storage class specifiers.

Ans : (Nov.-20, Dec.-18, Dec.-16, Imp.)

Refer Unit-III, Q.No. 16

5. Write a short notes on Recursion Functions.

Ans : (Nov.-20, June-19, Imp.)

Refer Unit-III, Q.No. 19

6. What are pointers ? Explain in detail with examples ?

Ans : (June-19, Dec.-18, June-17, Imp.)

Refer Unit-III, Q.No. 22

7. Explain array of pointers with example ? (or) Explain about passing array to functions.

Ans : (June-19, June-18, Imp.)

Refer Unit-III, Q.No. 29

8. Explain pointers with character strings ? (or) Explain about pointers and strings.

Ans : (June-18)

Refer Unit-III, Q.No. 30

9. What is DMA ? Explain the related functions with examples ?

Ans : (Nov.-20, (Dec.-17, June-17, Imp.)

Refer Unit-III, Q.No. 15

UNIT - IV

1. What are structures ? Explain.

Ans : (Nov.-20, June-19, Dec.-17, Imp.)

Refer Unit-IV, Q.No. 1

2. Explain declaration, initialization & accessing of structure and its members with example.

Ans : (Nov.-20, June-19, Dec.-18, Dec.-16, Imp.)

Refer Unit-IV, Q.No. 2.

3. Explain different ways to Initialization of structure.

Ans : (Dec.-16)

Refer Unit-IV, Q.No. 3

4. Explain the process of Accessing Members of a Structure.

Ans : (Nov.-20, Dec.-16, Imp.)

Refer Unit-IV, Q.No. 4

5. What are the differences between structures & unions ?

Ans : (June-18, Dec.-16, Imp.)

Refer Unit-IV, Q.No. 10

6. What are enumerated data types?

Ans : (Dec.-18)

Refer Unit-IV, Q.No. 11

7. What are files ? Explain its types ?

Ans : (June-19, Dec.-17)

Refer Unit-IV, Q.No. 12

8. Explain about Opening ,Reading, Writing and closing a file ?

Ans : (June-18, June-17, Imp.)

Refer Unit-IV, Q.No. 18

9. Explain the functions used in reading and writing of text files ?

Ans : (Dec.-19, Dec.-18, Dec.-16, Imp.)

Refer Unit-IV, Q.No. 19

10. Explain the functions to access Random files in C ?

Ans : (June-18, June-17)

Refer Unit-IV, Q.No. 21

11. Explain about file management functions and operations ?

Ans : (Dec.-18, June-18, June-17)

Refer Unit-IV, Q.No. 22

UNIT I

Computer Fundamentals: Introduction of Computers, Classification of Computers, Anatomy of a Computer, Memory Hierarchy, Introduction to OS, Operational Overview of a CPU. Program fundamentals: Generation and Classification of Programming Languages, Compiling, Interpreting, Loading, Linking of a Program, Developing Program, Software Development. Algorithms: Definitions, Different Ways of Stating Algorithms (Step-form, Pseudo-code,

Flowchart), Strategy for Designing Algorithms, Structured Programming Concept. Basics of C: Overview of C, Developing Programs in C, Parts of Simple C Program, Structure of a C Program, Comments, Program Statements, C Tokens, Keywords, Identifiers, Data Types, Variables, Constants, Operators and Expressions, Expression Evaluation-precedence and associativity, Type Conversions.

1.1 INTRODUCTION TO COMPUTERS

Q1. What is a computer ? Explain.

Ans : (Dec.-16)

Computers have touched every part of our lives: the way we work, the way we learn, the way we live, even the way we play. It almost is impossible to go through a single day without encountering a computer, a device dependent on a computer, information produced by a computer, or a word that was introduced or whose meaning has changed with the advent of computers. Because of the significance of computers in today's world, it is important to be computer literate. Being computer literate means you have knowledge and understanding of computers and their uses.

Computer is a Powerful Tool

A computer is a powerful tool because it is able to perform the information processing cycle operations (input, process, output, and storage) with amazing speed, reliability, and accuracy; store huge amounts of data and information; and communicate with other computers. Computers allow users to generate correct information quickly, hold the information so it is available at any time, and share the information with other computer users.

Definition

A **computer** is a machine that manipulates data according to a list of instructions.

Computer is a modern electronic tool that accepts input data, do processing according to a set of instructions, called program and produce/store

the output information. The term computer has been derived from the word 'COMPUTE'. This computer generally gives impression that it is a calculating machine but really computer can do several jobs other than computation.

In general, the data and information are often used interchangeably. In particular, in IT, the data and information are two distinct objects and one should clearly understand the difference between them. **Data** represents raw facts and observation. Examples are number, word, amount, name concept, holiday, etc. **Information** is the result of processing of the data. That means, the data that has been transformed in useful and meaningful form is known as information.

A computer system is a combination of **hardware** and **software**, which are used for processing and storing the data. Hardware refers to the physical components of a computer system. Examples are microprocessor, keyboard, Cathode Ray Tube (CRT), harddisk, floppy disk, mouse etc., Software is the set of instructions (called program) written for a computer to perform a particular task.

The two principal characteristics of a computer are :

- It responds to a specific set of instructions in a well-defined manner.
- It can execute a prerecorded list of instructions (a program).

Modern computers are electronic and digital. The actual machinery – wires, transistors, and circuits – is called hardware; the instructions and data are called software.

All general-purpose computers require the following hardware components:

- **Memory** : Enables a computer to store, at least temporarily, data and programs.
- **Mass storage device** : Allows a computer to permanently retain large amounts of data. Common mass storage devices include disk drives and tape drives.
- **Input device** : Usually a keyboard and mouse, the input device is the conduit through which data and instructions enter a computer.
- **Output device** : A display screen, printer, or other device that lets you see what the computer has accomplished.
- **Central processing unit (CPU)** : The heart of the computer, this is the component that actually executes instructions.

Capabilities of a Computer

Computers are versatile tools with seemingly limitless capabilities. These capabilities are categorized depending on their type and application. Some of the common capabilities of computers are given below :

Computers are capable of :

1. Performing tasks (or computations) at very high speed.
2. Performing computations with desired accuracy.
3. Receiving the data and producing useful information.
4. Storing instructions and data for future processing or use.
5. Executing a set of instructions (called program) with or without manual intervention in a well-defined manner.
6. Supporting decision making by using logical rules.
7. Communicating with other systems, even those at remote places.

8. Exploiting a complex internal structure in a variety of ways.

9. Performing the tasks (or subtasks) repetitively.

Q2. List the characteristics of computer ? Explain.

Ans :

Today, computers are every where - in our offices, homes, appliances, automobiles - the list is endless. Much of the world runs on computers, and computers have profoundly changed our lives, mostly for the better. Let us discuss some of the characteristics of a computer, which makes them an essential part of every emerging technology and such a desirable tool in human development.

1. Automatic

An automatic machine works by itself without human intervention. Computers are automatic machines because once started on a job, they carry out the job (normally without any human assistance) until it is finished. However, computers being machines cannot start themselves and cannot go out and find their own problems and solutions. We need to instruct a computer using coded instructions that specify exactly how it will do a particular job. Some of the other characteristics of computers (such as speed and accuracy) are because they are automatic and work on a problem without any human intervention.

2. Speed

A computer is a very fast device. It can perform in a few seconds, the amount of work that a human being can do in an entire year – if he/she worked day and night and did nothing else. In other words, a computer can do in a few minutes what would take a man his entire lifetime.

While talking about the speed of a computer we do not talk in terms of seconds or even milliseconds (10^{-3}) but in terms of

microseconds (10^{-6}), nanoseconds (10^{-9}), and even picoseconds (10^{-12}). A powerful computer is capable of performing several billion (10^9) simple arithmetic operations per second.

3. Accuracy

In addition to being very fast, computers are very accurate. Accuracy of a computer is consistently high and the degree of its accuracy depends upon its design. A computer performs every calculation with the same accuracy.

However, errors can occur in a computer. These errors are mainly due to human rather than technological weakness. For example, errors may occur due to imprecise thinking by a programmer or incorrect input data. We often refer to computer errors caused due to incorrect input data or unreliable programs as garbage-in-garbage-out (GIGO).

4. Diligence

Unlike human beings, a computer is free from monotony, tiredness, and lack of concentration. It can continuously work for hours without creating any error and without grumbling. Hence, computers score over human beings in doing routine type of jobs that require great accuracy. If ten million calculations have to be performed, a computer will perform the last one with exactly the same accuracy and speed as the first one.

5. Versatility

Versatility is one of the most wonderful things about a computer. One moment it is preparing results of an examination, next moment it is busy preparing electricity bills, and in between, it may be helping an office secretary to trace an important letter in seconds. All that is required to change its talent is to slip in a new program into it. In brief, a computer is capable of performing almost any task can be reduced to a finite series of logical steps.

6. Power of Remembering

As a human being acquires new knowledge, his/her brain subconsciously selects what it feels to be important and worth retaining in memory. The brain relegates unimportant details to back of mind or just forgets them. This is not the case with computers.

A computer can store and recall any amount of information because of its secondary storage (a type of detachable memory) capability. It can retain a piece of information as long as a user desires and the user can recall the information whenever required. A computer forgets or losses certain information only when a user asks it to do so. Hence, it is entirely up to the user to make a computer retain or forget some information.

7. No I.Q.

A computer is not a magical device. It possesses no intelligence of its own. Its I.Q. is zero, at least until today. It has to be told what to do and in what sequence. Hence, only a user determines what tasks a computer will perform. A computer cannot take its own decision in this regard.

Limitations of Computers

Although computers are very fast and accurate still they have some limitations :

1. Computer cannot think on its own. It processes the data according to the specifications given by the user. A computer cannot handle errors by itself. Software has to be developed so as to handle both physical and logical errors.
2. Computers can process towards a goal only when all the steps are properly specified. The task has to be defined in detail.
3. They cannot take decisions on their own. They cannot draw any logical deductions without going through all the steps.
4. Unlike human beings they cannot learn from the past experience. The operational logic has to be provided through command instructions and programs in order to make them work.

Q3. Discuss the generations of computers?

Ans :

In computer technology, technological advancement will be measured by generations. A specific system or software or language is said to belong to a specific 'generation'. Each generation indicates a significant change in computer design. The revolution in the electronic industry started from the first generation (1950s) and now we are moving towards the fifth generation.

First Generation (1942-1955)

A vacuum tube was a fragile glass device, which used filaments as a source of electronics and could control and amplify electronic signals. It was the only high-speed electronic switching device available in those days. These vacuum tube computers could perform computations in milliseconds and were referred to as first-generation computers.

The memory of these computers used electromagnetic relays, and all data and instructions were fed into the system from punched cards.

Features of first-generation computers are as follows :

1. They were the fastest calculating devices of their time.
2. They were too bulky in size, requiring large rooms for installation.
3. They used thousands of vacuum tubes that emitted large amount of heat and burnt out frequently. Hence, the room/ areas in which these computers were located had to be properly air-conditioned.
4. Each vacuum tube consumed about half a watt of power. Since a computer typically used more than ten thousand vacuum tubes, power consumption of these computers was very high.
5. As vacuum tubes used filaments, they had a limited life. Because a computer used thousands of vacuum tubes, these computers were prone to frequent hardware failures.

6. Due to low mean time between failures, these computers required constant maintenance.
7. In these computers, thousands of individual components were assembled manually by hand into electronic circuits. Hence, commercial production of these computers was difficult and costly.
8. Since these computers were difficult to program and use, they had limited commercial use.

Second Generation (1955-1964)

John Bardeen, Willian Shockley, and Walter Brattain invented a new electronic switching device called transistor at Bell Laboratories in 1947. Transistors soon proved to be a better electronic switching device than vacuum tubes due to their following properties :

1. They were more rugged and easier to handle than tubes since they were made of germanium semiconductor materials rather than glass.
2. They were highly reliable as compared to tubes since they had no parts like a filament that could burn out.
3. They could switch much faster than tubes. Hence, switching circuits made of transistors could operate much faster than their counterparts made of tubes.
4. They consumed almost one-tenth the power consumed by a tube.
5. They were much smaller than a tube.
6. They were less expensive to produce.
7. They dissipated much less heat as compared to vacuum tubes.

The second-generation computers were manufactured using transistors instead of vacuum tubes. Due to the properties of transistors listed above, these computers were more powerful, more reliable, less expensive, smaller, and cooler to operate than the first-generation computers. They used magnetic cores for main memory, and magnetic disk and tape as secondary storage media. Punched cards were still popular and widely used for preparing and feeding programs and data to these computers.

High-level programming languages (like FORTRAN, COBOL, ALGOL, and SNOBOL) and batch operating system emerged during second generation. High-level programming languages were easier for people to understand and work with than assembly or machine languages making second-generation computers easier to program and use than first-generation computers.

The first-generation computers were mainly used for scientific computations. However, the second-generation computers were increasingly used in business and industry for commercial data processing applications like payroll, inventory control, marketing, and production planning.

Features of second-generation computers are as follows :

1. They were more than ten times faster than the first-generation computers.
2. They were smaller than first-generation computers and required smaller space.
3. They consumed less power and dissipated less heat than the first-generation computers. The rooms/areas in which the second-generation computers were located still required to be properly air-conditioned.
4. They were more reliable and less prone to hardware failures than the first-generation computers.
5. They had faster and larger primary and secondary storage as compared to first-generation computers.
6. They were easier to program and use than the first-generation computers. Hence, they had wider commercial use.
7. In these computers, thousands of individual transistors had to be assembled manually by hand into electronic circuits making commercial production of these computers difficult and costly.

Third Generation (1964-1975)

Computers built using integrated circuits characterized the third generation. Integrated circuits (called ICs) are circuits consisting of several electronic components like transistors, resistors, and

capacitors grown on a single chip of silicon eliminating wired interconnection between components. The IC technology was also known as “microelectronics” technology because it made it possible to integrate larger number of circuit components into very small (less than 5 mm square) surface of silicon, known as “chip”. Initially the integrated circuits contained only about ten to twenty components. This technology was named small scale integration (SSI). The advancement in technology for manufacturing ICs, it became possible to integrate up to about hundred components on a single chip. This technology came to be known as medium scale integration (MSI).

ICs were smaller, less expensive to produce, more rugged and reliable, faster in operation, dissipated less heat, and consumed less power than circuits built by wiring electronic components manually. Hence, third-generation computers were more powerful, more reliable, less expensive, smaller and cooler to operate than second-generation computers.

Parallel advancements in storage technologies allowed construction of larger magnetic core based random access memory as well as larger capacity magnetic disks and tapes. Hence, third-generation computers typically had few megabytes (less than 5 Megabytes) of main memory and magnetic disks capable of storing few tens of megabytes of data per disk drive.

Standardization of high-level programming languages, timesharing operating systems, unbundling of software from hardware, and creation of an independent software industry happened during third generation. FORTRAN and COBOL were the most popular high-level programming languages in those days. American National Standards Institute (ANSI) standardized them in 1966 and 1968 respectively, and the standardized versions were called ANSI FORTRAN and ANSI COBOL.

Features of third-generation computers are as follows :

1. They were more powerful than second-generation computers. They were capable of performing about 1 million instructions per second.

2. They were smaller than second-generation computers requiring smaller space.
3. They consumed less power and dissipated less heat than second-generation computers. The rooms/areas in which third-generation computers were located still required to be properly air-conditioned.
4. They were more reliable and less prone to hardware failures than second-generation computers requiring lower maintenance cost.
5. They had faster and larger primary and secondary storage as compared to second-generation computers.
6. They were general-purpose machines suitable for both scientific and commercial applications.
7. Their manufacturing did not require manual assembly of individual components into electronic circuits resulting in reduced human labor and cost involved at assembly stage. Hence, commercial production of these systems was easier and cheaper. However, highly sophisticated technology and expensive setup was required for the manufacture to IC chips.
8. Standardization of high-level programming languages allowed programs written for one computer to be easily ported to and executed on another computer.
9. Timesharing operating system allowed interactive usage and simultaneous use of these systems by multiple users.
10. Timesharing operating system helped in drastically improving the productivity of programmers cutting down the time and cost of program development by several fold.
11. Timesharing operating system also made on-line systems feasible resulting in usage of these systems for new on-line applications.
12. Minicomputers of third-generation made computers affordable even by smaller companies.

Fourth Generation (1975-1989)

Average number of electronic components packed on a silicon chip doubled each year after 1965. This progress soon led to the era of large

scale integration (LSI) when it was possible to integrate over 30,000 electronic components on a single chip, followed by very large scale integration (VLSI) when it was possible to integrate about one million electronic components on a single chip.

A microprocessor contains all circuits needed to perform arithmetic logic and control functions, the core activities of all computers, on a single chip. Hence, it became possible to build a complete computer with a microprocessor, a few additional primary storage chips, and other support circuitry. It started a new social revolution – personal computer (PC) revolution.

Several new developments emerged to match the new technologies of fourth generation. For example, several new operating systems were developed for PCs. Notable among these were MS-DOS, MS-Windows, and Apple's propriety OS. Since PCs were for individuals who were not computer professionals, companies developed graphical user interfaces for making computers more user friendly. A graphical user interface (GUI) provides icons (pictures) and menus (list of choices) that users can select with a mouse. This enables new computer users to learn to use a PC quickly. During fourth-generation period, UNIX operating system and C programming language also became very popular.

Features of fourth-generation computers are as follows :

1. PCs were smaller and cheaper than mainframes or minicomputers of third generation.
2. Although the fourth-generation mainframes required proper air-conditioning of the rooms/areas in which they were located, no air-conditioning was required for PCs.
3. They consumed less power than third-generation computers.
4. They were more reliable and less prone to hardware failures than third-generation computers requiring negligible maintenance cost.
5. They had faster and larger primary and secondary storage as compared to third-generation computers.

6. They were general-purpose machines.
7. Their manufacturing did not require manual assembly of individual components into electronic circuits resulting in reduced human labor and cost involved at assembly stage. Hence, commercial production of these systems was easier and cheaper.
8. Use of standard high-level programming languages allowed programs written for one computer to be easily ported to and executed on another computer.
9. Graphical user interface (GUI) enabled new users to quickly learn how to use computers.
10. PC-based applications made PCs a powerful tool for both office and home usage.
11. Network of computers enabled sharing of resources like disks, printers, etc. among multiple computers and their users. They also enabled several new types of applications involving interaction among computer users at geographically distinct locations.
12. In addition to unbundled software, these systems also used add-on hardware feature that allowed users to invest only in the hardware configuration and software of their need and value.
13. PCs of fourth generation made computers affordable even by individuals for their personal use at home.

Fifth Generation (1989 - Present)

VLSI technology became ULSI (Ultra Large Scale Integration) technology in fifth generation resulting in production of microprocessor chips having ten million electronic components. The speed of microprocessors and the size of main memory and hard disk doubled almost every eighteen months.

Storage technology also advanced very fast making larger and larger main memory and disk storage available in newly introduced systems. During fifth generation, optical disks also emerged as a popular portable mass storage media such as CD-ROM.

Communication technologies became faster day-by-day and more and more computers were networked together. This trend resulted in emergence and popularity of the Internet and associated technologies and applications. The Internet made it possible for computer users sitting across the globe to communicate with each other within minutes by use of electronic mail (known as *e-mail*) facility. A vast ocean of information became readily available to computer users through the World Wide Web (WWW).

Features of fifth-generation computers are as follows :

1. Portable PCs (notebook computers) are much smaller and handy than PCs of fourth generation allowing users to use computing facility even while travelling.
2. Fifth generation desktop PCs and workstations are several times more powerful than PCs of fourth generation.
3. Although fifth generation mainframes require proper air-conditioning of the rooms/areas in which they are located, no air-conditioning if normally required for notebook computers, desktop PCs, and workstations.
4. They consume less power than their predecessors do.
5. They are more reliable and less prone to hardware failures than their predecessors were, requiring negligible maintenance cost.
6. Many of the large-scale fifth-generation systems have hot-plug feature that enables a failed component to be replaced with a new one without the need to shutdown the system. Hence, the uptime of these systems is very high.
7. They have faster and larger primary and secondary storage as compared to their predecessors. They are general-purpose machines.
8. Use of standard high-level programming languages allows programs written for one computer to be easily ported to and executed on another computer.
9. Newer and more powerful applications, including multimedia applications, make the systems more useful in every occupation.

Q4. List the applications of computers?

Ans :

For the last few decades, computer technology has revolutionized the business and other aspects of human life all over the world. Every company, large or small, is now directly or indirectly dependent on computers for information processing. Computer systems also help in the efficient operations of railway and airway reservation, hospital records, accounts, electronic banking, and so on. Computers not only save time, but also save paper work.

Some of the areas where computers are being used are listed below :

1. Education

Computers have also revolutionized the whole process of education. Currently, the classrooms, libraries, and museums are efficiently utilizing computers to make the education much more interesting. Unlike recorded television shows, computer-aided education (CAE) and Computer Based Training (CBT) packages are making learning much more interactive.

2. Science

Scientists have been using computers to develop theories, to analyse, and test the data. The fast speed and the accuracy of the computer allow different scientific analyses to be carried out. They can be used to generate detailed studies of how earthquakes affect buildings or pollution affects weather pattern. Satellite-based applications have not been possible without the use of computers. It would also not be possible to get the information of our solar system and the cosmos without computers.

3. Medicine and Health Care

There has been an increasing use of computers in the field of medicine. Now, doctors are using computers right from diagnosing the illness to monitoring a patient's status during complex surgery. By using automated imaging techniques, doctors are able to look inside a person's body and can study each organ in detail (such as CAT scans and MRI scans), which was not possible few years ago. There are several examples of special purpose computers that can operate within the human body such as a cochlear implant, a special kind of hearing aid that makes it possible for deaf people to hear.

4. Engineering/Architecture/Manufacturing

The architects and engineers are extensively using computers in designing and drawings. Computers can create objects that can be viewed from all the three dimensions. By using the techniques like virtual reality, architects can explore houses that have been designed but not built. Besides, Computer Aided Manufacturing (CAM) can be used in designing the product, ordering the parts, and planning production. Thus, computers help in coordinating the entire manufacturing processes.

5. Communication

E-mail or electronic mail is one of the communication media in which computer is used. Through e-mail the messages and reports are passed from one person to one or more persons by the aid of computer and the telephone line. Advantage of this service is that while transferring the messages it saves time, avoids wastage of paper. The person who is receiving the messages can read the messages whenever he is free and can save it, reply it, forward it or delete it from the computer.

6. Entertainment

Computers are finding greater use in entertainment industry. Computers are used to control the images and sounds. The special effects, which mesmerize the audience, would not have been possible without the computers. Computerised animation and colorful graphics have modernized the film industry.

7. Business Application

This is one of the important uses of the computer. Initially computers were used for batch processing jobs, where one does not require the immediate response from the computer. Currently computers are mainly used for real time applications that require immediate response from the computer. There are various concerns for which computers are used such as in business forecasting, to prepare pay bills and personal records, in banking operations and data storage, in the various types of Life Insurance Business, and as an aid to management.

Businesses are also using the networking of computers, where a number of computers are connected together to share the data and the information. Use of e-mail and Internet has changed the ways of business.

8. Publishing

Computers have created a field known as DTP or Desk Top Publishing. In DTP, with the help of computer and a laser printer one can perform the publishing job all by oneself. Many of the tasks requiring long manual hours such as making table of contents and index can be automatically done by the application of computers and DTP software.

9. Banking

In the field of banking and finance, computers are extensively used. People can use the ATM services 24 hours of the day in order to deposit and withdraw cash. When the different branches of the bank are connected through the computer networks, then the inter branch transaction such as cheque and draft can be done by the computers without any delay.

1.2 CLASSIFICATION OF COMPUTERS**Q5. Classify different types of computers on size and types?**

Ans : (Dec.-19, Dec.-16)

Computers differ based on their data processing abilities. They are classified according to purpose, data handling and functionality.

According to purpose, computers are either general purpose or specific purpose. General purpose computers are designed to perform a range of tasks. They have the ability to store numerous programs, but lack in speed and efficiency.

Specific purpose computers are designed to handle a specific problem or to perform a specific task. A set of instructions is built into the machine.

Classification based on Operating Principles

Based on the operating principles, computers can be classified into one of the following types :-

1. Digital Computers
2. Analog Computers
3. Hybrid Computers

1. Digital Computers

Operate essentially by counting. All quantities are expressed as discrete or numbers. Digital computers are useful for evaluating arithmetic expressions and manipulations of data (such as preparation of bills, ledgers, solution of simultaneous equations etc).

Digital Computers operate on inputs which are ON-OFF type and its output is also in the form of ON-OFF signal. Normally, an ON is represented by a 1 and an OFF is represented by a 0. So we can say that digital computers process information which is based on the presence or the absence of an electrical charge or we prefer to say a binary 1 or 0.



A digital computer can be used to process numeric as well as non-numeric data. It can perform arithmetic operations like addition, subtraction, multiplication and division and also logical operations. Most of the computers available today are digital computers. The most common examples of digital computers are accounting machines and calculators.

The results of digital computers are more accurate than the results of analog computers. Analog computers are faster than digital. Analog computers lack memory whereas digital computers store information.

2. Analog Computers

Analog computers are used to process analog data. Analog data is of continuous nature and which is not discrete or separate. Such type of data includes temperature, pressure, speed weight, voltage, depth etc. These quantities are continuous and having an infinite variety of values.

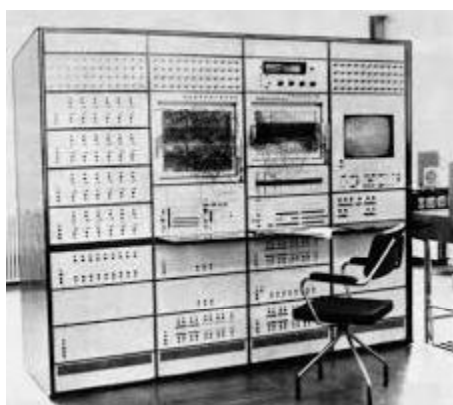


It measures continuous changes in some physical quantity e.g. The Speedometer of a car measures speed, the change of temperature is measured by a Thermometer, the weight is measured by Weights machine. These computers are ideal in situations where data can be accepted directly from measuring instrument without having to convert it into numbers or codes.

Analog computers are the first computers being developed and provided the basis for the development of the modern digital computers. Analog computers are widely used for certain specialized engineering and scientific applications, for calculation and measurement of analog quantities. Analog computers do not require any storage capability because they measure and compare quantities in a single operation. Output from an analog computer is generally in the form of readings on a series of dial (Speedometer of a car) or a graph on strip chart.

3. Hybrid Computers

A hybrid is a combination of digital and analog computers. It combines the best features of both types of computers, i.e., It has the speed of analog computer and the memory and accuracy of digital computer.



Hybrid computers are used mainly in specialized applications where both kinds of data need to be processed. Therefore, they help the user, to process both continuous and discrete data. For example a petrol pump contains a processor that converts fuel flow measurements into quantity and price values.

In hospital Intensive Care Unit (ICU), an analog device is used which measures patient's blood pressure and temperature etc, which are then converted and displayed in the form of digits. Hybrid computers for example are used for scientific calculations, in defense and radar systems.

Classification of Computers based on area of applications

Modern computers depending upon their applications are classified as: -

- a) Special Purpose Computers
- b) General Purpose Computers

a) Special Purpose Computers

A special purpose computer is designed only to meet the requirements of a particular task or application. The instructions needed to perform a particular task are permanently stored into the internal memory, so that it can perform the given task on a single command. It therefore doesn't possess unnecessary options and is less expensive.

b) General Purpose Computers

A General Purpose computers are designed to meet the needs of many different applications. In these computers, the instructions needed to perform a particular task are wired permanently into the internal memory. When one job is over, instructions for another job can be loaded into the internal memory for processing. This, a general purpose machine can be used to prepare pay-bills, manage inventories, print sales report and so on.

Classification of Computers based on size and Capability

Based on size and capability, computers are broadly classified into

1. Microcomputers (Personal Computer)

A microcomputer is the smallest general purpose processing system. The older pc started 8 bit processor with speed of 3.7MB and current pc 64 bit processor with speed of 4.66 GB.

Examples : IBM PCs, APPLE computers

Microcomputer can be classified into 2 types

1. **Desktops :** A desktop computer is a personal computer designed for regular use at a single location on or near a desk or table due to its size and power requirements.
2. **Portables :** A portable computer is a computer that is designed to be moved from one place to another and includes a display and keyboard.

The difference is portables can be used while travelling whereas desktops computers cannot be carried around.

The different portable computers are: -

1. Laptop
2. Notebooks
3. Palmtop (hand held)
4. Wearable computers

Laptop computer is similar to a desktop computers but the size is smaller. They are expensive than desktop. The weight of laptop is around 3 to 5 kg. A laptop combines the components, inputs, outputs, and capabilities of a desktop computer, including the display screen, speakers, a keyboard, pointing devices (such as a touchpad or trackpad), a processor, and memory into a single unit.

A **notebook computer** is a battery- or AC-powered personal computer generally smaller than a briefcase that can easily be transported and conveniently used in temporary spaces such as on airplanes, in libraries, temporary offices, and at meetings. A notebook computer, sometimes called a laptop computer, typically weighs less than 5 pounds and is 3 inches or less in thickness. Among the best-known makers of notebook and laptop computers are IBM, Apple, Compaq, Dell, Toshiba, and Hewlett-Packard.

A **palmtop computer** is a personal computer or other electronic device that has many of the same features as a computer and fits in the palm of your hand. Because of their size early palmtop computers did not have a keyboard or a mouse and often relied

off a pen that used Graffiti or something similar. Palmtop computers were discontinued in late 2000's as all the manufacturers shifted to tablets.

A **wearable computer** is a digital device that is either strapped to or carried on a user's body. It is used most often in research that focuses on behavioral modeling, health monitoring systems, IT and media development, where the person wearing the computer actually moves or is otherwise engaged with his or her surroundings.

Wearable computers provide constant computer and user interaction. In extreme cases, they serve much like a prosthetic, in that device use does not require users to cease other activities. Sony produced a wristwatch called Sony Smart Watch that must be paired with an Android phone. Once paired, it becomes an additional remote display and notification tool.

2. Minicomputers

A minicomputer is a type of computer that possesses most of the features and capabilities of a large computer but is smaller in physical size.

A minicomputer fills the space between the mainframe and microcomputer, and is smaller than the former but larger than the latter. Minicomputers are mainly used as small or midrange servers operating business and scientific applications. However, the use of the term minicomputer has diminished and has merged with servers. A minicomputer may also be called a mid-range computer.

3. Mainframe computers

Mainframes are a type of computer that generally are known for their large size, amount of storage, processing power and high level of reliability. They are primarily used by large organizations for mission-critical applications requiring high volumes of data processing. In general, there are a few characteristics of mainframes that are common among all mainframe vendors: Nearly all mainframes have the ability to run (or host) multiple operating systems. Mainframes can add or hot swap system capacity without disruption. Mainframes are

designed to handle very high volume input and output (I/O) and emphasize throughput computing. A single mainframe can replace dozens or even hundreds of smaller servers.



4. Supercomputers

Supercomputers have extremely large storage capacity and computing speeds which are many times faster than other computers. A supercomputer is measured in terms of tens of millions Instructions per second (mips), an operation is made up of numerous instructions. The supercomputer is mainly used for large scale numerical problems in scientific and engineering disciplines such as Weather analysis.



Classification based on number of microprocessors

Based on the number of microprocessors, computers can be classified into

- (a) Sequential computers and
- (b) Parallel computers

(a) Sequential Computers

Any task complete in sequential computers is with one microcomputer only. Most of the computers (today) we see are sequential computers where in any task is completed sequentially instruction after instruction from the beginning to the end.

(b) Parallel Computers

The parallel computer is relatively fast. New types of computers that use a large number of processors. The processors perform different tasks independently and simultaneously thus improving the speed of execution of complex programs dramatically. Parallel computers match the speed of supercomputers at a fraction of the cost.

1.3 ANATOMY OF A COMPUTER**Q6. Explain the Anatomy of a Computer**

Ans : (June-18)

A computer can accept input, process or store data, and produce output according to a set of instructions which are fed into it. The two main characteristics of a computer are: (i) it responds to a specific set of instructions in a well-defined manner. (ii) It can execute a pre-recorded list of instructions (a program). Modern computers are electronic and digital. The actual machinery - wires, transistors and circuits is called hardware; the instructions and data are called software.

All general purpose computers require the following hardware components :

- i) Central Processing Unit (CPU):** The 'brain' of the computer, the component that actually executes instructions.
- ii) Memory :** It enables a computer to store, at least temporarily, data and programs.
- iii) Input device :** Usually a keyboard or mouse is used to read data and programs into the computer.
- iv) Output device :** A display screen, printer, etc. that lets you see what the computer has accomplished.

- v) Mass storage device :** It allows a computer to permanently store large amounts of data. Common mass storage devices include disk drive and tape drive.

Q7. What is CPU? Explain?

Ans :

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

A CPU is a complex integrated circuit made of silicon. A modern CPU can have billions of microscopic transistors mounted on it. The CPU is typically enclosed by a metallic heat spreader which allows for better heat dissipation.

1. Control Unit (CU)

The control unit tells the rest of the computer system how to carry out a program's instructions. It directs the movement of electronic signals between memory - which temporarily holds data, instructions and processes information - and the ALU. It also directs these control signals between the CPU and input/output devices.

2. Arithmetic - Logic Unit (ALU)

Arithmetic Logic Unit, usually called the ALU, performs two types of operation - arithmetical and logical. Arithmetical operations are the fundamental mathematical operations consisting of addition, subtraction, multiplication and division. Logical operations consist of comparisons. That is two pieces of data are compared to see whether one is equal to, less than, or greater than the other.

3. Registers

Computers also have several additional storage locations called registers. These appear in the Control Unit and ALU and make

processing more efficient. Registers are a sort of special hi-speed storage areas that hold data and instructions temporarily during processing. They are parts of the Control Unit and ALU rather than the memory. Their contents can, therefore be handled much faster than the contents of the memory.

4. Addresses

To locate the characters of data or instructions in the main memory, the computer stores them in locations known as addresses. A unique number designates each address. Addresses can be compared to post office mailboxes. Their numbers remain the same, but contents continuously change.

Q8. Explain memory unit? its types.

Ans :

The processor performs all the fundamental computation of the computer system. Other components contribute to the computation by doing such things as storing data or moving data into and out of the processor. But the processor is where the fundamental action takes place.

A processor chip has relatively little memory. It has only enough memory to hold a few instructions of a program and the data they process. Complete programs and data sets are held in memory external to the processor. This memory is of two fundamental types: **main memory, and secondary memory.**

Main memory is sometimes called volatile because it loses its information when power is removed. Secondary memory is usually nonvolatile because it retains its information when power is removed. (However, it needs power when information is stored into memory or retrieved from it.)

Main memory is sometimes called main storage and secondary memory is sometimes called secondary storage or mass storage.

Main Memory

Main memory is where programs and data are kept when the processor is actively using them.

When programs and data become active, they are copied from secondary memory into main memory where the processor can interact with them. A copy remains in secondary memory.

Main memory is intimately connected to the processor, so moving instructions and data into and out of the processor is very fast.

Main memory is sometimes called **RAM**. RAM stands for **Random Access Memory**. "Random" means that the memory cells can be accessed in any order. However, properly speaking, "RAM" means the type of silicon chip used to implement main memory.

When people say that a computer has "512 megabytes of RAM" they are talking about how big its main memory is. One megabyte of memory is enough to hold approximately one million (10⁶) characters of a word processing document.

Nothing permanent is kept in main memory. Sometimes data are placed in main memory for just a few seconds, only as long as they are needed.

Secondary Memory

Secondary memory is where programs and data are kept on a long-term basis. Common secondary storage devices are the hard disk and optical disks.

- The hard disk has enormous storage capacity compared to main memory.
- The hard disk is usually contained inside the case of a computer.
- The hard disk is used for long-term storage of programs and data.
- Data and programs on the hard disk are organized into files.
- A file is a collection of data on the disk that has a name.

A hard disk might have a storage capacity of 500 gigabytes (room for about 500 x 10⁹ characters). This is about 100 times the capacity of main memory. A hard disk is slow compared to

main memory. If the disk were the only type of memory the computer system would slow down to a crawl. The reason for having two types of storage is this difference in speed and capacity.

Large blocks of data are copied from disk into main memory. The operation is slow, but lots of data is copied. Then the processor can quickly read and write small sections of that data in main memory. When it is done, a large block of data is written to disk.

Q9. What is interface? How is it related to memory?

Ans :

The interface unit inter connects the CPU with memory and also with the various input/output (I/O) devices. The instructions and data move between the CPU and other hardware components through interface unit.

It is a set of parallel wires or lines which connects all the internal computer components to the CPU and main memory. Depending on the type of data transmitted, a bus can be classified into the following three types :

i) Data bus

A data bus is a system within a computer or device, consisting of a connector or set of wires, that provides transportation for data. Different kinds of data buses have evolved along with personal computers and other pieces of hardware.

ii) Address Bus

An address bus is a computer bus architecture used to transfer data between devices that are identified by the hardware address of the physical memory (the physical address), which is stored in the form of binary numbers to enable the data bus to access memory storage.

The address bus carries the address of the piece of memory or I/O device to be read from or written to. It is a unidirectional bus, which is to say that data travels only one way; from the CPU to memory. The number of lines on the bus determines the number of addressable memory elements.

iii) Control Bus

A control bus is a computer bus that is used by the CPU to communicate with devices that are contained within the computer. This occurs through physical connections such as cables or printed circuits.

The CPU transmits a variety of control signals to components and devices to transmit control signals to the CPU using the control bus. One of the main objectives of a bus is to minimize the lines that are needed for communication.

Q10. What is software? Explain its categories?

Ans :

(June-17, Imp.)

Software is a set of programs, which is designed to perform a well-defined function. A program is a sequence of instructions written to solve a particular problem. There are two types of software :

1. System Software
2. Application Software

1. System Software

The system software is collection of programs designed to operate, control, and extend the processing capabilities of the computer itself. System software are generally prepared by computer manufactures. These software products comprise of programs written in low-level languages which interact with the

hardware at a very basic level. System software serves as the interface between hardware and the end users.

Examples of system software are Operating System, Compilers, Interpreter, Assemblers etc.

Features of system software are as follows :

- Close to system
- Fast in speed
- Difficult to design
- Difficult to understand
- Less interactive
- Smaller in size
- Difficult to manipulate
- Generally written in low-level language

2. Application Software

Application software products are designed to satisfy a particular need of a particular environment. All software applications prepared in the computer lab can come under the category of Application software.

Application software may consist of a single program, such as a Microsoft's notepad for writing and editing simple text. It may also consist of a collection of programs, often called a software package, which work together to accomplish a task, such as a spreadsheet package.

Examples

Application software are following :

- Payroll Software
- Student Record Software
- Inventory Management Software
- Income Tax Software
- Railways Reservation Software

- Microsoft Office Suite Software
- Microsoft Office

Features of application software are as follows:

- Close to user
- Easy to design
- More interactive
- Slow in speed
- Generally written in high-level language
- Easy to understand
- Easy to manipulate and use
- Bigger in size and requires large storage space.

1.4 MEMORY HIERARCHY

Q11. What is memory? Explain different types of memories in computer ?

Ans :

(Dec.-18, Dec.-17)

A memory is just like a human brain. It is used to store data and instructions. Computer memory is the storage space in computer where data is to be processed and instructions required for processing are stored. The memory is divided into large number of small parts called cells. Each location or cell has a unique address which varies from zero to memory size minus one.

For example if computer has 64k words, then this memory unit has $64 * 1024 = 65536$ memory locations. The address of these locations varies from 0 to 65535.

Memory is primarily of three types :

1. Cache Memory
2. Primary Memory/Main Memory
3. Secondary Memory

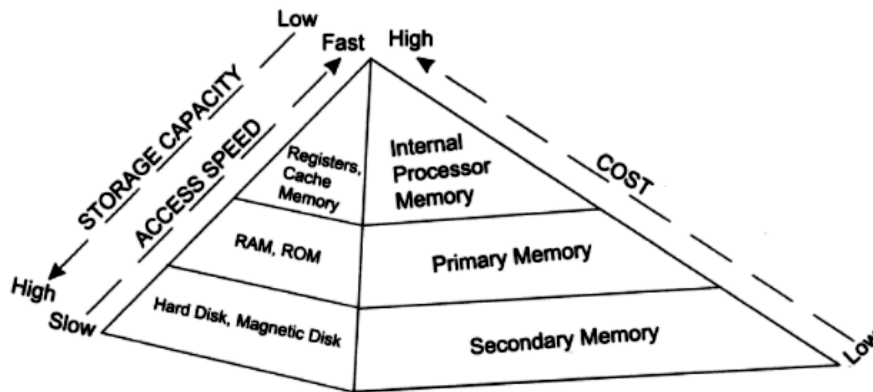


Fig.: Memory Hierarchy

Q12. What is cache memory? Explain its advantages & disadvantages.

Ans :

(Dec.-17)

Cache memory is a very high speed semiconductor memory which can speed up CPU. It acts as a buffer between the CPU and main memory. It is used to hold those parts of data and program which are most frequently used by CPU. The parts of data and programs are transferred from disk to cache memory by operating system, from where CPU can access them.

Advantages

- The advantages of cache memory are as follows:
- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

- The disadvantages of cache memory are as follows:
- Cache memory has limited capacity.
- It is very expensive.

Registers

Registers are part of the CPU. They have the least capacity, storing extremely limited amounts of instructions and data only immediately before and after processing.

Q13. Write a short notes on primary memory ?

Ans :

(Dec.-17)

The main memory or main storage of the computer is also known as **RAM** (Random Access Memory). It is constructed from integrated circuits and needs to have electrical power in order to maintain its information. When power is lost, the information is lost too! It can be directly accessed by the CPU. The

access time to read or write any particular byte are independent of whereabouts in the memory that byte is, and currently is approximately 50 **nanoseconds** (a thousand millionth of a second). This is broadly comparable with the speed at which the CPU will need to access data.

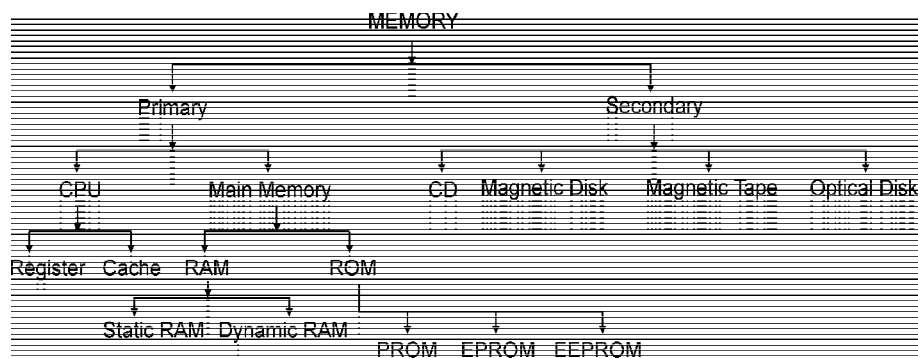


Figure: Classification of a Memory

Main memory is expensive compared to external memory so it has limited capacity. The capacity available for a given price is increasing all the time. For example many home Personal Computers now have a capacity of 16 **megabytes** (million bytes), while 64 megabytes is commonplace on commercial workstations. The CPU will normally transfer data to and from the main memory in groups of two, four or eight bytes, even if the operation it is undertaking only requires a single byte.

The terms main storage and auxiliary storage originated in the days of the mainframe computer to distinguish the more immediately accessible data storage from storage that required input/output operations. An earlier term for main storage was core in the days when the main data storage contained ferrite cores.

Random Access Memory

Random-access memory (usually known by its acronym, **RAM**) is a computer data storage. Today it takes the form of integrated circuits that allow the stored data to be accessed in any order, i.e. at random. The word random thus refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether or not it is related to the previous piece of data

RAM is temporary and volatile; that is RAM chips lose their contents if the current is lost or turned off. RAM chips are located directly on the computer's main circuit board or in other chips located on peripheral cards that plug into the main circuit board.

The two main types of RAM are dynamic RAM (DRAM) and static RAM (SRAM). DRAM memory chips offer the greatest capacities and the lowest cost per bit, but are relatively slow. SRAM costs more than DRAM but has a higher level of performance, making SRAM the preferred choice for performance-sensitive applications, including the external L2 and L3 caches that speed up microprocessor performance.

Read-only Memory

Read-only memory (ROM) is a class of storage media used in computers and other electronic devices. Because data stored in ROM cannot be modified (at least not very quickly or easily), it is mainly used to distribute firmware (software that is very closely tied to specific hardware, and unlikely to require frequent updates).

ROM refers only to **mask ROM** (the oldest type of solid state ROM), which is fabricated with the desired data permanently stored in it, and thus can never be modified. However, more modern types such as EPROM and flash EEPROM can be erased and reprogrammed multiple times; they are still described as “read-only memory” because the reprogramming process is generally infrequent, comparatively slow, and often does not permit random access writes to individual memory locations.

Types of ROM

- **Programmable read-only memory (PROM)**, or **onetime programmable ROM (OTP)**, can be written to or **programmed** via a special device called a **PROM programmer**. Typically, this device uses high voltages to permanently destroy or create internal links (fuses or antifuses) within the chip. Consequently, a PROM can only be programmed once.
- **Erasable programmable read-only memory (EPROM)** can be erased by exposure to strong ultraviolet light (typically for 10 minutes or longer), then rewritten with a process that again requires application of higher than usual voltage.
- **Electrically erasable programmable read-only memory (EEPROM)** is based on a similar semiconductor structure to EPROM, but allows its entire contents (or selected **banks**) to be electrically erased, then rewritten electrically, so that they need not be removed from the computer (or camera, MP3 player, etc.). Writing or **flashing** an EEPROM is much slower (milliseconds per bit) than reading from a ROM or writing to a RAM (nanoseconds in both cases), since available densities are not as great and the cost per bit is higher.
- **Flash memory** – Another form of rewritable ROM storage is called flash memory. This technology can be built into a system or installed on a personal computer card (known as a *flash card*). Flash memory via flash cards is very popular for small portable technologies such as cellular telephones, digital cameras, handheld computers, and other consumer products.

Q14. What is a secondary memory? List explain same secondary storage devices?

Ans :

(Dec.-17)

Secondary storage is designed to store very large amounts of data for extended periods of time. Secondary storage can have memory capacity of several terabytes or more and only small portions of that data are placed in primary storage at any one time.

Secondary Storage has the Following Characteristics

- It is nonvolatile. It takes much more time to retrieve data from secondary storage than it does from RAM because of the electromechanical nature of secondary storage devices.
- It is much more cost effective than primary storage. It can take place on a variety of media each with its own technology.
- The overall trends in secondary storage are towards more direct-access methods, higher capacity with lower costs, and increased portability.

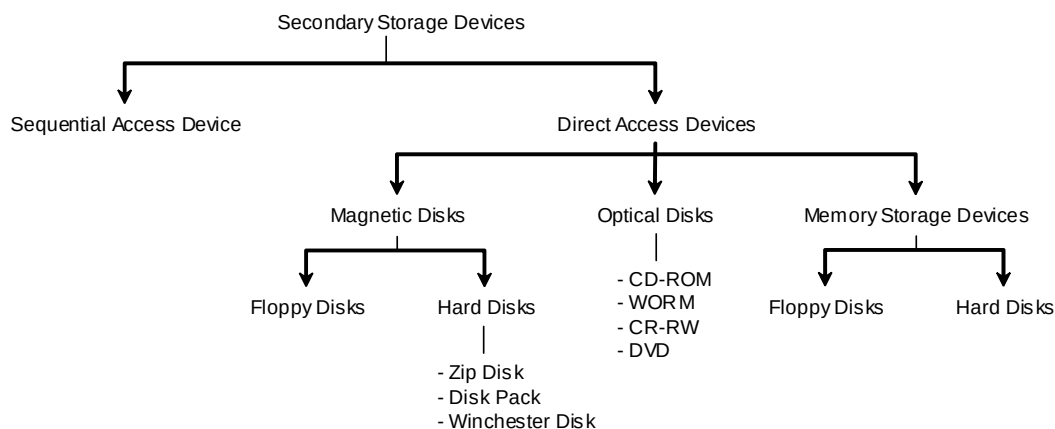


Fig.: Classification of Secondary storage devices

Figure above shows the popular ones used in today's computer systems. They are magnetic tape, magnetic disk, optical disk and memory storage devices.

Sequential and Direct-Access Devices

There are two methods of accessing information : sequential or serial access, and direct or random access. A sequential-access storage device is one in which the arrival at desired location may be preceded by sequencing through other locations so that access time varies according to location. That is, information on a sequential-access device can be retrieved in the same sequence only in which it is stored.

If we use a sequential-access device for data storage, the time taken to access the desired information may be long enough to cause frustration to the customer. Secondary storage devices exist that permit access to individual information in a more direct or immediate manner. These direct-access devices are also called random-access devices because out of all the information stored, any randomly chosen piece of information can be accessed immediately. Hence, a random-access storage device is one in which any location in the device may be selected at random, access to the information stored in direct, and approximately equal access time is required for each location.

Magnetic Disks

Magnetic disks are the most popular direct-access secondary storage device. They are also the most popular online secondary storage device.

A magnetic disk is a thin circular plate/platter made of metal or plastic and coated on both sides with a recording material that can be magnetized such as iron oxide. Data are recorded on a disk in the form of tiny invisible magnetized and non-magnetized spots (1s and 0s) on the coated surfaces of the disk. A standard binary code, usually 8-bit EBCDIC, is used for recording data.

Magnetic disks can also be erased and reused indefinitely. Old data on a disk are erased automatically by recording new data in the same area. However, stored data can be read many times without affecting the data.

All types of magnetic disks are broadly classified into two types – **floppy disks** and **hard disks**. Floppy disks are made of flexible plastic whereas, hard disks are made rigid metal. Depending on the type of packaging hard disks are further classified into zip disks, disk packs and Winchester disks.

Hard Disks

A hard disk drive (HDD), hard disk, hard drive or fixed disk is a data storage device used for storing and retrieving digital information using one or more rigid rapidly rotating disks (platters) coated with magnetic material. The platters are paired with magnetic heads arranged on a moving actuator arm, which read and write data to the platter surfaces. Data is accessed in a random-access manner, meaning that individual blocks of data can be stored or retrieved in any order and not only sequentially. HDDs are a type of non-volatile memory, retaining stored data even when powered off.

Optical Disks

An optical disc is any media read using a laser assembly. The most common types of optical media are Blu-ray, CDs, and DVDs. Computers can read and write to CDs and DVDs using a CD Writer or DVD Writer drive, and a Blu-ray is read with a Blu-ray drive. Drives such as a CD-R and DVD-R drive that can read and write information to discs are known as magneto-optic (MO).

There are three main types of optical media: CD, DVD, and Blu-ray disc. CDs can store up to 700 megabytes (MB) of data and DVDs can store up to 8.4 GB of data. Blu-ray discs, which are the newest type of optical media, can store up to 50 GB of data.

1.5 OPERATING SYSTEMS

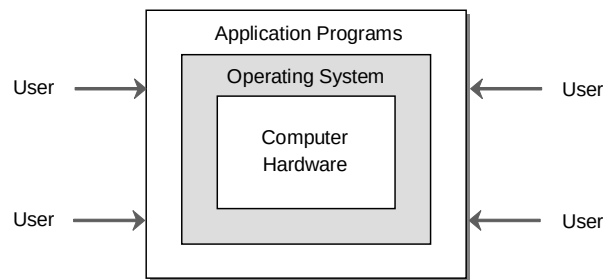
1.5.1 Introduction to Operating Systems

Q15. Define operating system ? Explain list the components of OS.

Ans : (June-17)

An operating system is an interface between hardware and user. An OS is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system acts as a host for computing applications run on the machine.

An operating system is a set of programs which act as an interface between the user of computer and the computer hardware, users or applications programs use the hardware indirectly through operating system as shown in figure.



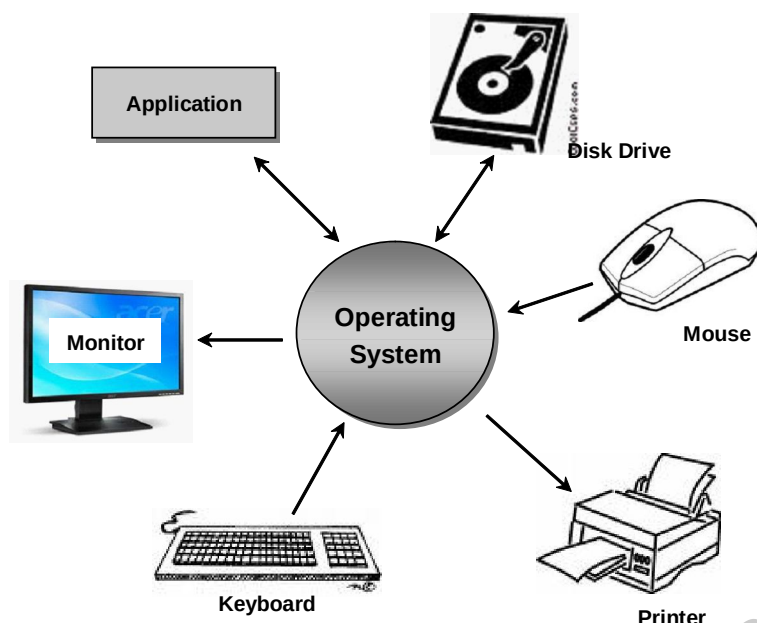
The operating system controls and coordinates the use of hardware among the application programs. The primary aim of an operating system is to provide convenience to the user in using the system, and the secondary aim is to use the system hardware in an efficient way. The OS is used to manage the various resources and operations of the computer system.

Operating systems can be classified as follows :

- 1. Multi-user :** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- 2. Multiprocessing :** Supports running a program on more than one CPU.
- 3. Multitasking :** Allows more than one program to run concurrently.
- 4. Multithreading :** Allows different parts of a single program to run concurrently.
- 5. Real time:** Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

Operating systems provide a software platform on top of which other programs, called application programs, can run. The application programs must be written to run on top of a particular operating system.

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.



Components of the Operating System

The operating system comprises a set of software packages that can be used to manage interactions with the hardware. The following elements are generally included in this set of software :

1. The **kernel**, which represents the operating system's basic functions such as management of memory, processes, files, main inputs/outputs and communication functionalities.
2. The **shell**, allowing communication with the operating system via a control language, letting the user control the peripherals without knowing the characteristics of the hardware used, management of physical addresses, etc.
3. The **file system**, allowing files to be recorded in a tree structure.

Functions of Operating System

An operating system (OS) is a set of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system.

Today most operating systems perform the following important functions :

1. **Program Creation:** The operating system provide editors, debuggers to assist the programmer in creating programs.
2. **Input / Output Operation :** A running program may require input and output. This I/O may involve a file or an I/O device. A user program cannot execute I/O operations directly. Thus operating system provide some means to do so.
3. **Program Execution:** A number of tasks required to execute a program. The task include instructions and data must be loaded into main memory. I/O devices and files must be initialized and other resources must be prepared. The operating system handles these tasks for the user.
4. **Error Detection :** The operating system detects the different types of errors and should take appropriate action. The errors include memory error, power failure, printer out of paper, illegal instruction in program.

5. **Resource Allocation :** The operating system collects the all the resources in the network environment or a system and grant these resources to requesting process. The resources include CPU cycles. Main memory, I/O devices and so on.
6. **Accounting:** The operating system can keep track of which users use now much and what kind of computer resources.
7. **Protection :** The operating system provides security Mechanism to protect the unauthorized usage of files in the network environment.
8. **Communication :** These are many situation in which one process needs to exchange information with another process. This exchange of information between two processes is communication. The communication may be implemented by technique like shared memory.
9. **Memory Management:** That is allocation of main memory and other storage areas to the system programmes as well as user programmes and data.
10. **File Management:** That is, the storage of file of various storage devices to another. It also allows all files to be easily changed and modified through the use of text editors or some other files manipulation routines.

1.5.2 Types of Operating System

Q16. Explain different types of operating systems?

Ans :

Within the broad family of operating systems, there are generally four types, categorized based on the types of computers they control and the sort of applications they support. The categories are :

1. Multiprogramming
2. Multiprocessing
3. Multitasking
4. Real-time Operating System

1. Multiprogramming

The central processing unit (CPU) is very fast, but input/output (I/O) devices are slow. Multiprogramming is a technique developed to minimize the effect of this mismatch between the CPU and I/O devices so that the CPU time is more efficiently used.

Commercial applications are I/O bound programs. Scientific and engineering programs involve very little I/O operations and large amount of computations, that is, these are the CPU bound programs. I/O bound operations are very slow. Separate I/O processors manage I/O operations initiated by the CPU. Multiprogramming increases CPU utilization by scheduling jobs for the CPU, so that it has a job to execute all the time.

The multiprogramming requires more sophisticated operating systems. The operating system keeps several programs in memory simultaneously to have several jobs in ready state for the CPU to execute. The operating system requires better memory management, job scheduling and CPU scheduling facilities. So multiprogramming requires large memories, fast secondary storage devices and fast ALU.

The multiprogramming operating system runs one program at a time. When that program requires some I/O operation, which is a slow process, the CPU switches to another job for execution when the first program execution waits for I/O operation to complete. If the second job also requires I/O operation, the CPU is allotted a third job and so on.

Multiprogramming is introduced in operating systems to overcome the problem of under-utilization of the CPU and main memory. Multiprogramming is the interleaved execution of two or more separate and independent programs by the same computer. The CPU switches from one job to another almost instantaneously. At any time only one program has control of the CPU. Thus the CPU is almost always busy. The memory is partitioned for several programs to reside in memory.

2. Multiprocessing

Multiprocessing systems have two or more CPUs, which have the ability to execute two or more jobs simultaneously. Instructions from several and independent programs are carried out by different CPUs at the same time. Multiprocessing systems can be designed in limitless number of ways. In some systems several small CPUs are interlinked to perform the major processing.

If one of the CPUs fail, another takes over and carries on with the processing. In some others a network is formed by interconnecting the CPUs and small CPUs, called front- end processors, are used for I/O operations, scheduling and controlling of jobs, etc. In yet other system, each CPU performs only specific type of applications.

Multiprogramming is similar to time sharing environment in which different applications share a single CPU. In time sharing, like multiprogramming, the CPU is not dedicated to a single user. The CPU switched between users so fast that the users do not feel that the CPU is switching between users. The multiple users can run multiple applications on a single CPU computer simultaneously. Multiprocessing systems have several CPUs.

Advantages of Multiprocessing

1. High reliability of the system is ensured by the provision of automatic takeover of complete work by other CPUs in case of breakdown of one of the CPUs.
2. Parallel processing improves systems efficiency by increasing throughput and lowering turnaround time of the system.
3. It also results in more efficient use of hardware resources other than the CPU.

Limitations

1. Large memory is required for multiprocessing as it requires many programs and data to reside in memory at any time.
2. The system needs a highly sophisticated operating system and highly skilled computer professionals to design it.

3. The multiprocessing systems are very expensive. It requires not only high initial outlay but also high operation and maintenance expenses are to be incurred.

3. Multitasking or Time Sharing

Multitasking is one such improvement and move towards making computers operate more efficiently and accomplishing more tasks in a given period of time. This technology aims at making devices independent to do their own processing to a great extent, without depending on the CPU for the processing. The objective was also to keep most of the hardwares operating most of the possible time, so that one can best use the available hardware devices in a given period of time.

The basic working methodology of this technique is that the devices work on their own most of the time, and whenever the CPU is required it is given a signal known as interrupt, which the CPU attends by keeping its current activity aside, then it returns to whatever task it was doing when the interrupt occurred. So, that the CPU can carry out multiple tasks simultaneously. This technology was originally used for mini and mainframe computers.

Multitasking is essential for multi-user or time sharing systems where several users share the same computer at the same time. The number of applications that can be run simultaneously depends on the available memory and the amount of processing required by each application. If too many applications are run concurrently, it slows down the response. If memory and processing power are adequate, multitasking permits users to switch from application to application and run application in background mode; for example printing in background.

Advantages of Time Sharing

1. Results in efficient use of software by allowing access to the same software for many users of the system.
2. It utilizes the CPU more efficiently by reducing CPU idle time.
3. Offers computing facility to many users simultaneously.
4. It provides quick response.

Limitations

1. Requires data communication facilities, as this time-shared system will have remote terminals using it.
2. Since many users use the system simultaneously, security and integrity of user programs and data becomes a problem.
3. The time-shared system supports several users at the same time operating even from remote terminals. Therefore its reliability must be ensured by back up facilities as a safety measure which can be used in the event of any system problem.

1.6 OPERATIONAL OVERVIEW OF A CPU

Q17. Define CPU ? Explain the operations of CPU.

Ans : (Dec.-17)

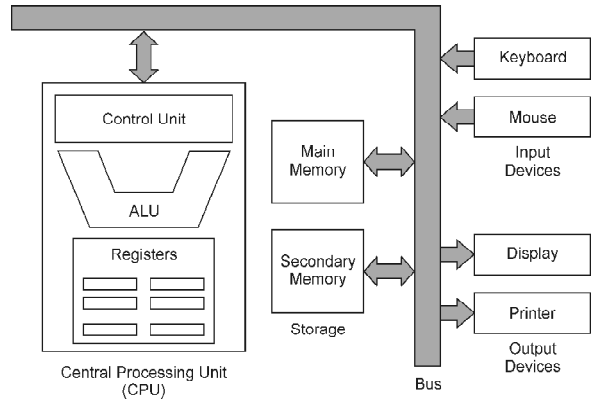
Central Processing Unit

Control unit (CU) and arithmetic logic unit (ALU) of a computer system are together known as the Central Processing Unit (CPU). The CPU is the brain of a computer system. In a human body, the brain takes all major decisions and other parts of the body function as directed by the brain. Similarly, in a computer system, all major calculations and comparisons take place inside the CPU and the CPU is responsible for activating and controlling the operations of other units of the computer system. The primary function of CPU is to execute the programs stored in the main memory.

CPU executes the programs by performing the following operations :

1. **Fetch instructions :** The CPU reads instructions from memory.
2. **Interpret instructions :** The fetched instruction is decoded to determine which action is needed.
3. **Fetch data :** CPU fetches the data either from the memory or from input device.
4. **Process data :** Performs arithmetic or logical operations on data.

- 5. Write results :** The results of an execution are sent either to a memory location or to an output device.



The main components of CPU are :

1. Arithmetic and logic unit (ALU)
2. Control Unit (CU)
3. Registers

1. Arithmetic and Logic Unit (ALU)

The ALU performs arithmetic and logical operations on data such as addition, subtraction, multiplication, division, comparison, AND, OR, NOT etc. ALU receives data as bits (a smallest unit of data) and executes the processing operations under the direction of program instructions. Most of the execution part within a computer is carried out by ALU.

2. Control Unit

The control unit manages and coordinates the entire computer system. It generates control and training signals necessary for the execution of instructions. This unit directs the specific instruction and data that are to be processed and determines the correct order and sequence. The control unit obtains instructions from the program stored in the main memory, interprets the instruction and issues signals that cause other units of the system to execute them.

3. Registers

Registers are temporary storage locations within the CPU where the data fetched from

memory can be held. Each register has a specific function. There is a specific register called the accumulator which holds one of the operands of operation to be performed in the ALU. The other registers often called general purpose registers are used to store the data and intermediate results during the execution of a program. In addition to these registers, CPU has a set of special purpose registers which are used for different purposes during program execution. Program counter, instructions register, memory address register, etc. are examples of special purpose registers.

CPU, memory and I/O devices are connected through a group of wires called a bus. These lines are meant to carry information. There are three types of buses : address bus, data bus and control bus. An address bus carries the address of memory location or an I/O device that the CPU wants to access. The address bus is unidirectional. The data bus carries data between memory and the CPU, and between CPU and I/O devices. The third bus, the control bus, carries the control and timing signals that coordinates the activities of all the components in the computer. The data and the control buses are bidirectional.

The function of a computer can be described as follows :

1. It accepts data and instructions through an input device and store them in the memory.
2. Information stored in the memory is fetched under program control, into the registers of CPU
3. ALU processes the data
4. All these activities inside the computer are controlled and directed by control and timing unit.

Control Unit

The control unit of CPU controls all computer operations. Basically the responsibility of a control unit is to control input/output operations, data transfers between different units and to govern the internal operations of CPU.

To execute an instruction, the control unit fetches program instructions from the main memory and decodes it. Control unit interprets the decoded instruction and identifies the operation to be performed and address of the operands, if any. It generates the required control and timing signals and sends them to the appropriate components of the system. These signals are necessary for the proper execution of instructions and the control of peripheral devices.

The control unit directs the ALU to process the required operations. The instructions are executed sequentially until any instruction calls for a branch or a jump to a different point in the program. Data transfer operations within the CPU as well as between the CPU and its peripherals are also controlled by the control unit.

The control unit performs two basic operations while executing a program : Fetch and execute instructions. That is

1. It fetches the instruction and determines the operation required for its execution.
2. It enables the CPU to execute the various operations required for the execution of the instruction in proper sequence.

Arithmetic and Logic Unit (ALU)

The functions of ALU is to perform arithmetic and logical operations. The ALU does the actual processing under program control. It performs these operations on the data as a result of the execution of the decoded instruction. Since both arithmetic and logical operations are performed by this unit, it is referred to as arithmetic and logic unit. Generally, the ALU performs the following operations :

1. Arithmetic operations such a addition, subtraction, division, multiplication etc.
2. Logical operations such as AND, OR, NOT (complement) and Exclusive OR.
3. Register operations such as left shift, clear, load, move, etc.

In a typical computer, the ALU has the necessary electronic circuitry to perform binary addition and shift operations. From these capabilities, routines can be constructed to perform subtraction, multiplication, by repeated addition and division by repeated subtraction and shifts.

Generally ALU is not capable of performing the complete mathematical operations such as exponential, trigonometric functions, floating point operation, etc. These operations are performed by either software routine or by an arithmetic coprocessor.

Most of the operations during execution cycle of an instruction are carried out in ALU. Data are presented to the ALU by bringing the require operands into the registers of the CPU, and the result and take logical decisions if necessary.

The arithmetic and logic operations are carried out by means of Logic gates which are basic building blocks of logic circuits. The four basic logic gates are OR, AND, NOT and Exclusive OR.

1.7 PROGRAM FUNDAMENTALS

1.7.1 Generation of Programming Languages

Q18. What is a programming language? Explain their classifications?

Ans : (Nov.-20, June-19, Imp.)

A **programming language** is a formal computer language or constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.

Programming language is a machine-readable artificial language designed to express computations that can be performed by a machine, particularly a computer.

Many programming languages have some form of written specification of their syntax and semantics, since computers require precisely defined instructions. Some (such as C) are defined by a specification document (for example, an ISO Standard), while others (such as Perl) have a dominant implementation.

In the present scenario of study programming languages become a very wide area of engineering and research. As the involvement of computer, automation and robotics growing in our daily life, programming becomes highly required to control all of them. To control all of these systems and machines and take desired output by them skilled

programming languages is necessary. However the area of programming language become how much wide but it will be under one of the three category of programming languages.

Computer languages can be classified into two major categories : Low-level and high-level. Low-level languages can be further divided into machine language and assembly language.

The figure below shows classification of computer languages.

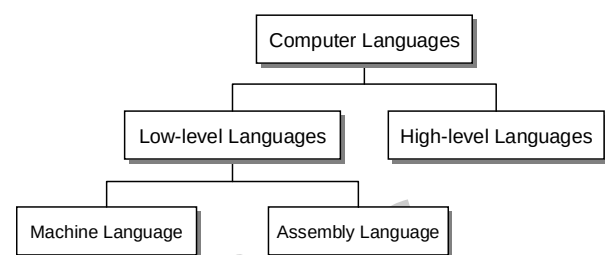
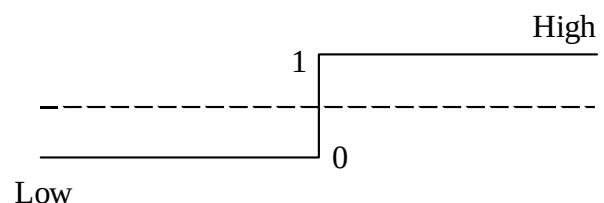


Fig.: Classification of Computer Languages

(i) Machine Languages (First Generation of Programming Languages)

Machine language is lowest level and named as the first generation of programming language. The fundamental method of representing information to computer is the electric pulse which is having states namely low and high. These two states can be associated with binary numbers 0 and 1 for convention.



A language (or code) in which the instructions are written as strings of binary numbers i.e., 0's and 1's is called **machine language** or **binary language**. Since the computer hardware recognizes (or works) binary states, it can only understand the binary language. Thus computer uses only the machine language when executing a program. There is a specific binary code for each instruction.

Instructions

Machine language instruction format consists two parts. OP code and operand.

OP CODE (OPERATION CODE)	OPERAND (ADDRESS/LOCATION)

Fig.: Instruction Format

OP code part specifies the function of the instruction that is to be performed by the computer. For example, the 00000001 OP code may specify the **addition** operation. Operand part specifies the address or location of operand on which the operation has to be performed. That is, provides an address to the CPU from where the data can be read or stored into.

(ii) Assembly Language (Second Generation of Programming Languages)

Machine language is the only language that the computer can understand and follow. It is made up to sequence of zeros and ones and is hence quite difficult for us to follow.

The **assembly languages** were developed in order to overcome the difficulties in writing machine language programs. Assembly language is middle level and named as the second generation of programming language.

In assembly language, each operation code of machine language is represented by a symbolic code called **mnemonic** and address of operand is represented by **symbolic address** instead of strings of 0's and 1's. For example, the registers and accumulator are all accessed by their names instead of their memory location. This makes the assembly language easier to read and understand. Assembly language is often referred to as **symbolic programming language**.

Each instruction of assembly language contains

- (a) Symbolic operation code and
- (b) Symbolic address

Symbolic codes are abbreviated words which are used in place of numeric (binary) **OP codes**.

The examples are :

ADD is mnemonic OP code for addition

SUB is mnemonic OP code for subtraction

LOAD is mnemonic OP code for load

STORE is mnemonic OP code for storing

Suppose to add two numbers stored in locations B and C and store the result in storage location D, the assembly language program for a hypothetical computer as

LOAD B Load the contents of B into accumulator

ADD C Add the contents of C to accumulator

STORE D store the result (contents of accumulator) into storage location D

There is a one-to-one correspondence between assembly language and machine language in the sense that for every symbolic instruction of assembly language, there is an equivalent binary instruction of machine language. Since instruction in both languages directly corresponds to a single machine function, these languages are called as **low-level languages**.

However, an assembly language program can not be directly executed on a computer because the computer understand machine language programs. Therefore, the program written using assembly language has to be translated into machine language. Such translation is performed by using software. A program which translates an assembly language program into machine language program is called an **Assembler**.

An assembler accepts a program written in assembly language, checks it and indicates the errors, if any, and generates a machine language equivalent of the original assembly language program.

Advantages of Assembly Language

1. No need of remembering all binary **OP codes** and **addresses** of memory location.
2. Program correction / modification is relatively easier

3. Less time is required in writing a program than machine language
4. Language is simple when compared with machine language.

Limitations of Assembly Language

1. Execution of program takes more time as it has to be converted into machine language.
2. Assembly language is machine dependent, i.e., it is designed for specific computer. Program has to be rewritten for a different machine.

(iii) High Level Languages (Third Generation of Programming Languages)

Low-level languages are machine oriented and machine dependent. Program development in low-level languages is very slow and tedious process. Programming, especially, in machine language is error prone. The limited features of low-level languages are overcome by the high-level languages.

High-level languages are machine independent as contrast to the low-level languages. That means, program written in such a language can usually run on computers of different types, sizes and manufacturers. The high-level languages are problem oriented languages in the sense that they are concerned with the problem to be solved rather than that of the machine.

The high-level language programs are very easy to write, modify and maintain. Also these programs are easy to portable from one system to another.

Advantages of High-level Languages

1. The high-level languages are easier to use and understand than the machine language and the assembly language.
2. A high-level language is more expressive than the low-level language and hence the programs written in high-level language are

much more compact and self explanatory than their low-level counterparts.

3. High-level languages provide a better or superior documentation than the low-level languages.
4. They are totally independent of the machine on which they are being used. This makes it possible for the users to use the same programs written on one machine, on different machine making the programs portable.
5. Correcting and modifying programs is much easier than low-level languages.

Limitations of High-level Languages

1. Less efficient as they take more execution time when compared to low-level languages.
2. The programmer cannot completely harness the total power available at hardware level.

1.7.2 Generations / Types of Programming Languages

Q19. Explain different types of programming languages ?

Ans : (Nov.-20, June-19, June-18, Imp.)

Procedural Languages

Procedural language is a type of computer programming language that specifies a series of well-structured steps and procedures within its programming context to compose a program. It contains a systematic order of statements, functions and commands to complete a computational task or program.

A procedural language is a computer programming language that follows, in order, a set of commands. Examples of computer procedural languages are BASIC, C, FORTRAN, and Pascal.

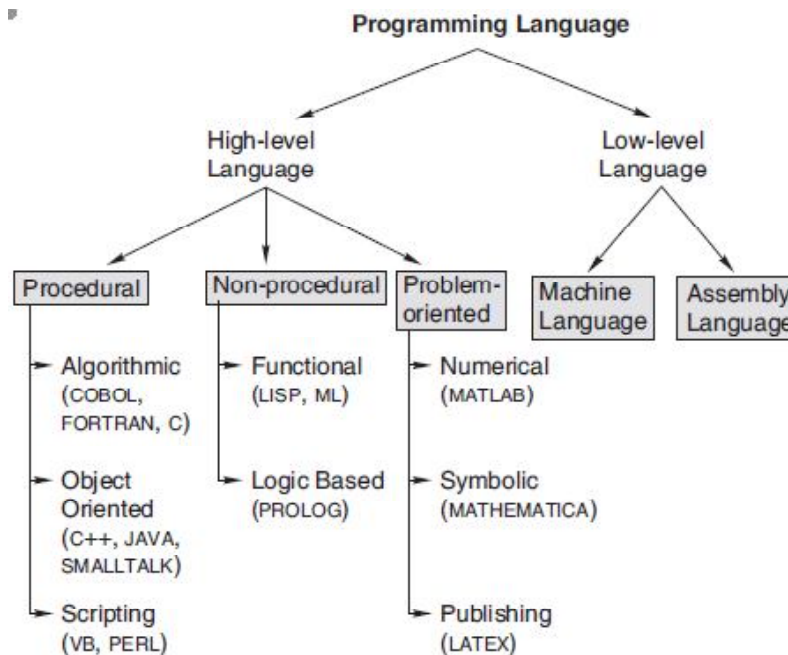


Fig.: Classification of programming languages

Algorithmic Language (ALGOL)

It is a family of portable programming languages for scientific computations that had a strong influence on other languages. These are high-level languages designed for forming convenient expression of procedures, used in the solution of a wide class of problems. In this language, the programmer must specify the steps the computer has to follow while executing a program.

In the sense that most modern languages are "Algol-like", it was arguably the most successful of the four high-level programming languages with which it was roughly contemporary : Fortran, Lisp, and COBOL. It was designed to avoid some of the perceived problems with FORTRAN and eventually gave rise to many other programming languages, including BCPL, B, Pascal, PL/I, Simula, and C.

Object-oriented Language

Object-oriented language is a high-level computer programming language that implements objects and their associated procedures within the programming context to create software programs.

Object-oriented language uses an object-oriented programming technique that binds related data and functions into an object and encourages reuse of these objects within the same and other programs. A good example of an object-oriented programming language is Java.

Objects are self-contained modules that contain data as well as the functions needed to manipulate the data within the same module. In a conventional programming language, data and subroutines or functions are separate. In object oriented programming, subroutines as well as data are locally defined in objects. The difference affects the way a programmer goes about writing a program as well as how information is represented and activated in the computer.

The most important object-oriented programming features are

- Encapsulation and data hiding
- Abstraction

- Polymorphism
- Inheritance
- Reusable code
- Message Communication
- Data abstraction

Scripting Languages

A script or scripting language is a computer language with a series of commands within a file that is capable of being executed without being compiled. Good examples of server side scripting languages include Perl, PHP, and Python. The best example of a client side scripting language is JavaScript.

Advantages of scripting languages :

- Open source, allowing users to view and edit the script if needed.
- Does not require the file to be compiled, but may be when necessary.
- Easy to learn and write.
- Easy to port between different operating systems.
- Much faster to develop than an actual program - some individuals and companies write scripts as a prototype for actual programs.

Problem-oriented Languages

Problem-oriented Languages were designed to solve specific problems e.g. querying databases and allowed the programmer to concentrate more on the problem rather than spending time learning the complex syntax of the language. Some of these fourth- generation languages are used for special applications as described below.

Query Languages enable users, who are not programmers, to use certain easily-understandable commands to search and generate reports from a database. An example would be the commands used by clerical officers to look for specific information on a personnel database-like date of first appointment or job title.

Application Generators contain a number of modules, which are logically related program statements that have been pre-programmed to

accomplish specific tasks. An example would be a module that calculates value-added tax (VAT) on products. The programmer can simply state which task is needed for a particular application. The application generator creates the program code by selecting the appropriate modules.

Non-Procedural Language

A programming language which does not require specific statements, instructions, or subroutines which need to be executed in a specified sequence.

Nonprocedural language is that in which a programmer can focus more on the code's conclusion and therefore doesn't have to use such common programming languages as JavaScript or C + +. This makes nonprocedural language somewhat easier to learn for those just beginning to get into coding and programming.

Nonprocedural language allows the programmer to focus on what to do rather than how to do it. This is because this language does not focus as much on individual processes needed to get to the conclusion but rather on ways to get to the conclusion itself. This is why nonprocedural language is often referred to as a "declarative language."

Functional Languages

These functional languages solve a problem by applying a set of functions to the initial variables in specific ways to get the answer. The functional programming style relies on the idea of function application rather than on the notion of variables and assignments. A program written in a functional language consists of function calls together with arguments to functions. LISP, ML, PERL, PYTHON etc. are examples of functional languages.

Logic-based programming language is a programming paradigm based on formal logic. A program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain. Logic programming languages are important because of their declarative nature, their potential power and flexibility, and their suitability for execution on highly parallel architectures. Major logic programming language families include Prolog, Answer set programming (ASP) and Datalog.

1.8 COMPILING, INTERPRETING, LOADING, LINKING OF A PROGRAM

Q20. Explain the following :

- (a) **Compiler**
- (b) **Interpreter**
- (c) **Assembler**
- (d) **Linker**
- (e) **Loader**

Ans :

(a) Compiler

A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program.

Compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. More generally, compilers are a specific type of translator.

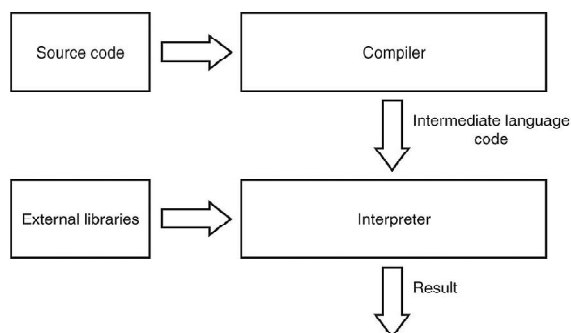


Fig.: Compiler action

During the process of translation, the compiler reads the source program statement-wise and checks for syntax errors. In case of any error, the computer generates a printout of the same. This action is known as diagnostics.

Structure of a Compiler

Compilers bridge source programs in high-level languages with the underlying hardware. A compiler verifies code syntax, generates efficient object code, performs run-time organization, and formats the output according to assembler and linker conventions.

A compiler consists of :

1. The Front End

Verifies syntax and semantics, and generates an intermediate representation or IR of the source code for processing by the middle-end. Performs type checking by collecting type information. Generates errors and warnings, if any, in a useful way. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis.

2. The Middle End

Performs optimizations, including removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. Generates another IR for the backend.

3. The Back End

Generates the assembly code, performing register allocation in process. Optimizes target code utilization of the hardware by figuring out how to keep parallel execution units busy, filling delay slots. Although most algorithms for optimization are NP-hard, heuristic techniques are well-developed.

(b) Interpreter

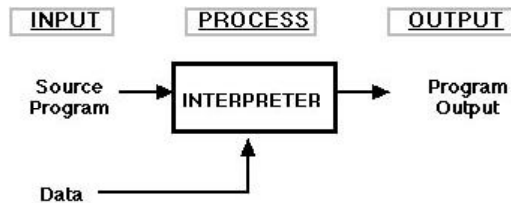
An interpreter is a computer program that directly executes, i.e. performs, instructions written in a programming or scripting language, without previously compiling them into a machine language program.

An interpreter generally uses one of the following strategies for program execution:

- Parse the source code and perform its behavior directly.
- Translate source code into some efficient intermediate representation and immediately execute this.

- Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.

An interpreter translates high-level instructions into an intermediate form, which it then executes. In contrast, a compiler translates high-level instructions directly into machine language. Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long. The interpreter, on the other hand, can immediately execute high-level programs.



Interpreters

- Interpreter takes single instruction as input .
- No intermediate object code is generated
- Conditional control statements are executes slower.
- Memory requirement is less
- Every time higher level program is converted into lower level program
- Errors are displayed for every instruction interpreted.
- The interpreter can immediately execute high-level programs, thus interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly.
- In addition, interpreters are often used in education because they allow students to program interactively.

The execution of a program written in high-level language involves the following steps :

1. Translation of the program resulting in the object program.
2. Linking of the translated program with other object programs needed for execution, thereby resulting in a binary program.
3. Relocation of the program to execute from the specific memory area allocated to it.
4. Loading of the program in the memory for the purpose of execution.

Compiler Vs Interpreter

Compiler	Interpreter
1. Compiler Takes Entire program as input.	1. Interpreter Takes Single instruction as input.
2. Intermediate Object Code is Generated	2. No Intermediate Object Code is Generated.
3. Conditional Control Statements are Executes faster.	3. Conditional Control Statements are Executes slower.
4. Memory requirement is more. (Since Object Code is Generated)	4. Memory requirement is less.
5. Program need not be compiled every time.	5. Every time higher level program is converted into lower level program.
6. Errors are displayed after entire program is checked.	6. Errors are displayed for every instruction interpreted (if any).
7. Example : C & C++	7. Example : BASIC, Python, Ruby

(c) Assembler

An assembler is a low-level programming language for a computer, or other programmable device, in which there is a very strong correspondence between the language and the architecture's machine code instructions. Each assembly language is specific to a particular computer architecture. In contrast, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling. Assembly language may also be called symbolic machine code.

Assembly language is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as assembly, or assembling the source code. Assembly time is the computational step where an assembler is run.

How Assembler Works

Most computers come with a specified set of very basic instructions that correspond to the basic machine operations that the computer can perform.

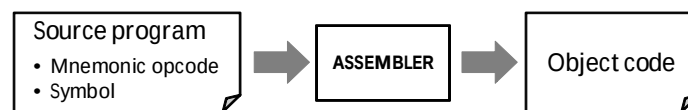


Fig.: Assembler

For example, a "Load" instruction causes the processor to move a string of bits from a location in the processor's memory to a special holding place called a register. Assuming the processor has at least eight registers, each numbered, the following instruction would move the value (string of bits of a certain length) at memory location 3000 into the holding place called register 8 :

L 8,3000

- The programmer can write a program using a sequence of these assembler instructions.
- This sequence of assembler instructions, known as the source code or source program, is then specified to the assembler program when that program is started.
- The assembler program takes each program statement in the source program and generates a corresponding bit stream or pattern (a series of 0's and 1's of a given length).
- The output of the assembler program is called the object code or object program relative to the input source program. The sequence of 0's and 1's that constitute the object program is sometimes called machine code.
- The object program can then be run (or executed) whenever desired.

(d) Linker

In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not built in libraries, it also links the user defined functions to the user defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

In C, files are translated separately. Thus, only function calls that cross file boundaries and references to global data require linking. Linking makes the addresses of programs known to each other so that transfer of control from one subprogram to another or a main program takes place during execution.

Relocation

Relocation is the process of assigning load addresses to various parts of a program and adjusting the code and data in the program to reflect the assigned addresses. A linker usually performs relocation in conjunction with symbol resolution, the process of searching files and libraries to replace symbolic references or names of libraries with actual usable addresses in memory before running a program.

Relocation is typically done by the linker at link time, but it can also be done at run time by a relocating loader, or by the running program itself. Some architectures avoid relocation entirely by deferring address assignment to run time; this is known as zero address arithmetic.

Relocation is typically done in two steps :

1. Each object file has various sections like code, data, .bss etc. To combine all the objects to a single executable, the linker merges all sections of similar type into a single section of that type. The linker then assigns run time addresses to each section and each symbol. At this point, the code (functions) and data (global variables) will have unique run time addresses.
2. Each section refers to one or more symbols which should be modified so that they point to the correct run time addresses based on information stored in a relocation table in the object file.

(e) Loader

Loader is a program that loads machine codes of a program into the system memory. In Computing, a loader is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

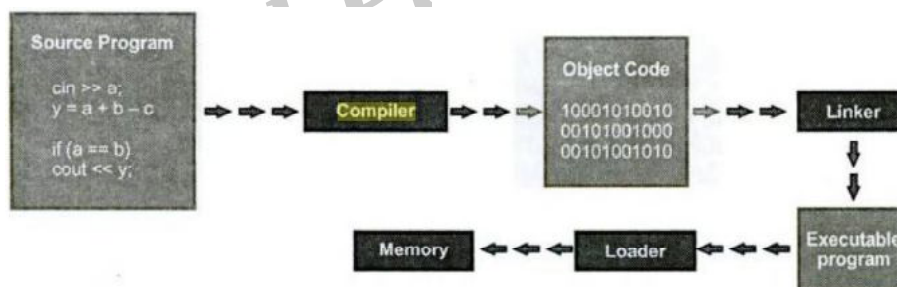


Fig.: Source Code Translation into Executable Code

A loader is a system program that accepts object programs and prepares them for execution and initiates the execution. The functions performed by the loader are :

- Assignment of load-time storage area to the program
- Loading of program into assigned area
- Relocation of program to execute properly from its load time storage area
- Linking of programs with one another

Thus, a loader is a program that places a program's instructions and data into primary storage locations. An absolute loader places these items into the precise locations indicated in the machine language program. A relocating loader may load a program at various places in primary storage depending on the availability of primary storage area at the time of loading. A program may be relocated dynamically with the help of a relocating register.

1.9 DEVELOPING A PROGRAM

Q21. Explain the process of developing a program?

Ans : (June-18)

A program consists of a series of instructions that a computer processes to perform the required operation. In addition, it also includes some fixed data, required to perform the instructions, and the process of defining those instructions and data. Thus, in order to design a program, a programmer must determine three basic principles :

1. The instructions to be performed.
2. The order in which those instructions are to be performed.
3. The data required to perform those instructions.

To perform a task using a program, a programmer has to consider various inputs of the program along with the process, which is required to convert the input into desired output.

Suppose we want to calculate the sum of two numbers, X and Y, and store the sum in Z, here X and Y are the inputs, addition is the process, and Z is the output of the program.

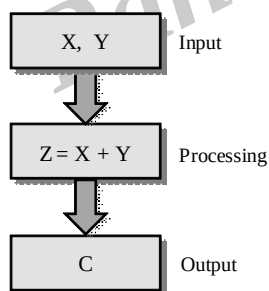


Fig.: Program performing a task

Program Development Cycle

'Program development cycle' is the technique which is commonly used for solving the problem. This technique uses number of steps. However, these steps may vary from person to person since the steps depend on the approach used by the person to solve the problem. Using this technique, the program development is carried out in steps. After completing these required steps, new demands

may appear and all the steps are repeated. Thus, often the process runs is a loop.

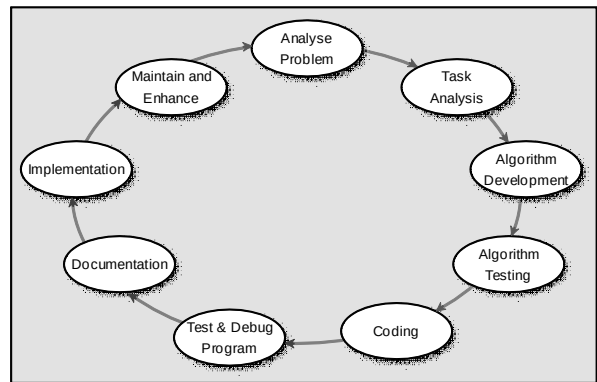


Fig.: Program development cycle

The steps included in the development cycle are :

1. Analyse or Define the Problem

An understanding of the problem is an important step. So, firstly the problem is analysed precisely and completely. According to this analysis, the developer comes to know the scope within which the development of the problem is to be done.

2. Task Analysis

After analysing the problem, the developer has to develop different solutions to solve the given problem. The same problem can have different solutions, but the optimum solution is to be selected. While selecting the solution, easiness and cost factors are considered.

3. Developing Algorithm

After choosing the appropriate solution, next step is developing an algorithm. That is, logical steps are depicted according to the selected solution. Then, the pictorial representation of an algorithm, that is flowchart is generated.

4. Testing the Algorithm for Accuracy

The accuracy of the algorithm should be checked before obtaining the actual code. During this checking, logical errors are identified. It is very difficult to detect and correct the logical errors at later stages. So it is better to do at early stage. The testing also makes sure that the algorithm is true.

5. Coding

This step involves actual coding of the program in the selected programming language. There are different levels of programming languages such as machine language, assembly language or high-level language. The selection of the language is done according to the available resources and application area.

6. Testing and Debugging the Program

Usually the initial program code contains errors. The syntax errors are found by a compiler and test data machine-designed by programmer. The results are also calculated manually. The results obtained are compared with results calculated manually from the test data. Several rounds of testing may be required.

7. Documentation

After removing all errors in the program, program developer has to make suitable documentation. These documents should be provided to the program users. This helps the user to operate the program correctly. It also can be used by other persons as a guide to understand the program clearly. The others can make corrections or modify the program if necessary.

8. Implementation

This step involves installing the program on the end user's machine. The user can understand the working of the program with the help of documents supplied along with the program. When the user uses the program, he comes to know the shortcomings or drawbacks of the program if any. The user informs this to the developer. The programmer can make changes or modify the program based on the feedback from the user.

9. Maintenance and Enhancement

After the program is implemented, it should be properly maintained taking care of the changing requirements of its users and system.

The program should be regularly enhanced by adding additional capabilities. This phase is also concerned with detecting and fixing the errors, which were missed in testing phase. Since this step generates user feedback, the programming cycle continues as the program is modified or reconstructed to meet the changing needs.

1.10 SOFTWARE DEVELOPMENT**Q22. Define software development ? Explain briefly about SDLC?**

Ans : (June-18)

Software development is the process of computer programming, documenting, testing, and bug fixing involved in creating and maintaining applications and frameworks resulting in a software product. Software development is a process of writing and maintaining the source code, but in a broader sense it includes all that is involved between the conception of the desired software through to the final manifestation of the software, sometimes in a planned and structured process.

Software development may include research, new development, prototyping, modification, reuse, re-engineering, maintenance, or any other activities that result in software products.

There are many approaches to software project management, known as software development life cycle models, methodologies, processes, or models. The waterfall model is a traditional version, contrasted with the more recent innovation of agile software development.

Software can be developed for a variety of purposes, the three most common being to meet specific needs of a specific client/business (the case with custom software), to meet a perceived need of some set of potential users (the case with commercial and open source software), or for personal use (e.g. a scientist may write software to automate a mundane task).

There are many approaches to software project management, known as software development life cycle models, methodologies, processes, or models. The waterfall model is a traditional version, contrasted with the more recent innovation of agile software development.

Software Development Life Cycle

SDLC, Software Development Life Cycle is a process used by software industry to design, develop and test high quality softwares. The SDLC aims to produce a high quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle and is also called as Software development process.
- The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process.
- ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.

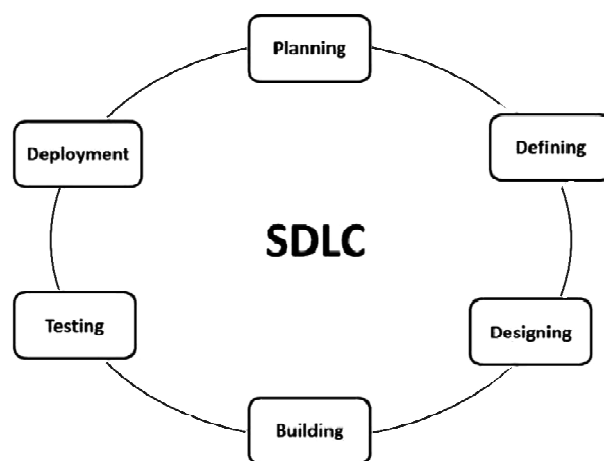


Fig.: SDLC

Q23. Explain different stages of SDLC in detail?

Ans :

A typical Software Development life cycle consists of the following stages:

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational, and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through SRS. Software Requirement Specification document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the product architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers have to follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers etc are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java, and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However this stage refers to the testing only stage of the product where products defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometime product deployment happens in stages as per the organizations. Business strategy. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

1.11 ALGORITHMS

1.11.1 Definition

Q24. What is an algorithm? Explain the characteristics & steps for writing algorithm ?

Ans : (Dec.-18, Dec.-17)

Algorithm is a finite sequence of explicit and unambiguous steps required to solve the given problem. When the required input values are provided, it produces an output and then terminates. The steps may repeat or require decisions as per the requirement.

The same problem can be solved with different methods. So, for solving the same problem, different algorithms may be accomplished. In these algorithms, the steps, time, space and efforts may vary more or less. User writes algorithm in his/her own language. So it cannot be executed on computer.

Characteristics of Algorithms

Algorithms are not computer programs, as they cannot be executed by a computer. The characteristics of a good algorithm are :

- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Input** – An algorithm has zero or more inputs, taken from a specified set of objects.
- **Output** – An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Generality** – the algorithm applies to a set of inputs.
- **Effectiveness** – All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

Example

Let us write an algorithm to calculate the addition of two given numbers.

Algorithm

Step 1 : Start

Step 2 : Read two numbers as X and Y.

Step 3 : Add numbers X and Y and store results as Z.

Step 4 : Display Z.

Step 5 : Stop

Thus, algorithm is the basic tool, used to develop the logic for solving the problem. Once the algorithm is ready, it indicates the solution of the problem is properly designed. After this the flow-chart is drawn, pseudo-code is obtained and then the logic is coded into a programming language.

Common Elements of Algorithm

1. **Acquire data (input):** Some means of reading values from an external source; most algorithms require data values to define the specific problem (e.g., coefficients of a polynomial).
2. **Computation:** Some means of performing arithmetic computations, comparisons, testing logical conditions, and so forth...
3. **Selection:** Some means of choosing among two or more possible courses of action, based upon initial data, user input and/or computed results.
4. **Iteration:** Some means of repeatedly executing a collection of instructions, for a fixed number of times or until some logical condition holds.
5. **Report results (output):** Some means of reporting computed results to the user, or requesting additional data from the user.

1.12 DIFFERENT WAYS OF STATING ALGORITHMS**Q25. Explain different types of representation of algorithms.**

Ans : (Dec.-18)

Algorithm may be represented in various ways. There are four ways of representing algorithms. These steps are as follows:

1. Step-form
2. Pseudo-code
3. Flowcharts

1. Step-form Algorithm

In the step form representation, the procedure of solving a problem is stated with written statements. Each statement solves a part of problem and these together completes the solution. The step-form uses just normal language to define each procedure. Every statement, that defines an action, is logically related of preceding statement.

Stepwise Refinement

1. Work out each and every detail for each small piece of manageable solution procedure:
Every input and output dealt with and the transformation algorithms implemented in each small piece of solution procedure, which is also known as process, is detailed. Even the interfacing details between each small procedure are worked out.
2. Decompose any solution procedure into further smaller pieces and iterate until the desired level of detail is achieved.
3. Group processes together which have some commonality.
4. Group variables together which have some appropriate commonality.

5. Test each small procedure for its detail and correctness and its interfacing with the other small procedures.

Walk through each of the small procedures to determine whether it satisfies the primary requirements and would deliver the appropriate outputs. Also, suitable tests have to be carried out to verify the interfacing between various procedures. Hence, the top-down approach starts with a big and hazy goal. It breaks the big goal into smaller components. These components are themselves broken down into smaller parts. This strategy continues until the designer reaches the stage where he or she has concrete steps that can actually be carried out.

Advantages

1. It is a step by step solution to a given problem, hence easy to understand.
2. It has got a definite procedure, which can be exacted within a set period of time.
3. It is very easy to draw a flowchart using algorithm.
4. It has got definite procedures to generate required output.
5. It is easy to debug as every step has got its own logical sequence.
6. It is independent of programming languages.

Disadvantages

1. It is time consuming.
2. In large algorithms, the flow of program control becomes difficult to track.
3. It is difficult to convert it into a flow chart and then in to a program.
4. Algorithm lack visual representation of programming constructs like flowcharts, thus understanding the logic becomes relatively difficult.

2. Pseudocode

The pseudo-code is a written form representation of the algorithm. However, it differs from the step form as it uses a restricted vocabulary to define its action of solving the problem. One problem with human language is that it can seem to be imprecise. But the pseudo-code, which is in human language, tends towards more precision by using a limited vocabulary.

3. Flowcharts

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. Each step in the sequence is noted within a diagram shape. Steps are linked by connecting lines and directional arrows. This allows anyone to view the flowchart and logically follow the process from beginning to end.

Q26. What are flowcharts ? Explain its types? What are the Symbols used in ? flowcharts

Ans : (June-18, Dec.-18, June-17, Imp.)

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. Each step in the sequence is noted within a diagram shape. Steps are linked by connecting lines and directional arrows. This allows anyone to view the flowchart and logically follow the process from beginning to end.

The flowchart has different shapes, there are two shapes: those with rounded ends represent the start and end points of the process and rectangles are used to show the interim steps. These shapes are known as flowchart symbols. Table below shows some standard symbols which are frequently required in the flowcharts.

Flowchart Symbols

A flowchart includes the following symbols :

➤ Terminal Symbol

It is used to indicate the point at which flowchart begins and ends and the words start or stop is inserted to identify the point.

➤ Input/Output Symbol

This symbol identifies the logical positioning of input and output operations. I/O symbol notes the entry of computer data and the printed output of information. The following figure depicts the input and output operation. The I/O symbol indicates an input/output operations. Input and output symbols can be associated with Read and print symbols.

➤ Processing Symbol

The processing related to all types of data is indicated by the processing symbol. This symbol defines computations. Moving data from a storage area to an output area or indicating the sorting and manipulation of data.

➤ Decision Symbol

Logical operations are indicated in flow charts using decision symbols. A decision symbol has two main components.

A question that defines the logical operation. The results of the decision (i.e., Yes/No). The question asked within the decision symbol indicates the comparison necessary to support processing.

The question is referred to as a conditional statement. The results of the comparison are given in terms of Yes or No responses. They are called as the conditional responses or branches of the decision, since the branch chosen is totally dependent upon the answer to the conditional statement.

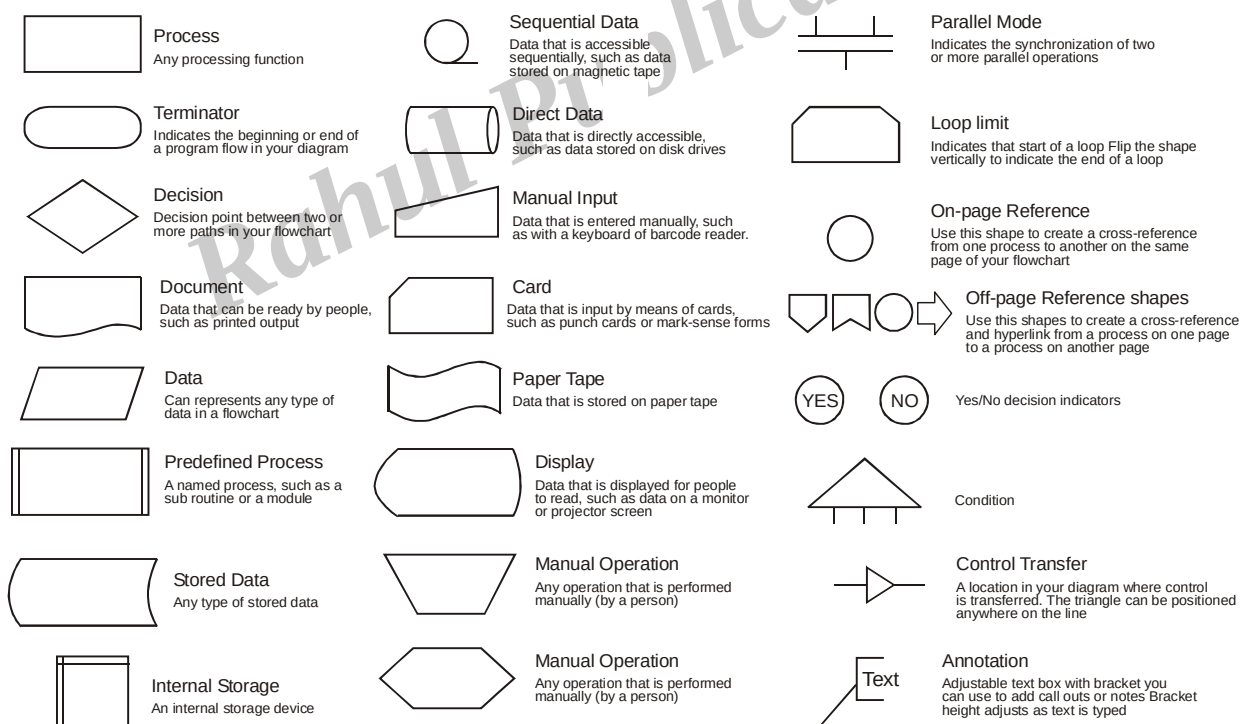


Fig.: Design Elements - Cross-Functional Flowcharts solution - Flowcharts Shapes

- **Annotation symbol :** It is used as vehicles to add descriptive comments or remarks to a flowchart. E.g., for condition statement

- **Flow lines :** Flow lines with over heads are used to indicate the flow of operation i.e. exact sequence in which the instructions are to be executed. The normal flow chart is from top to bottom and left to right.
- **Connector :** If a flowchart is very long, the flow lines start crossing at many places that causes confusion and reduces understandability of the flow chart. Sometimes a flowchart may be too long to fit in a single page and the use of flowlines becomes impossible.

Whenever such situation arises connectors are used as substitutes for flow lines. This symbol represents an entry from or an exit to, another part of the flowchart. A connector symbol is represented by a circle and a letter or digit is placed within the circle to represent the link.

Q27. List out Advantages and Limitations of Flowcharts?

Ans :

Advantages of Flowcharts

The flowchart outlines the logical sequence of a particular computer application and diagram order in which the data processing is to take place.

The benefits of using flowcharts are :

1. **Communication**
Flow charts are good visual aids for communicating the logic of a system to all concerned. They communicate the facts of a business problem to those whose skills are needed for arriving at the solution.
2. **Quicker Grasp of Relationships**
The flowcharts help in identifying the relationship existing among the problem elements. It helps the programmer to understand the current and proposed procedures more easily.
3. **Effective Analysis**
It becomes a model or blue print of programme or system that can be broken down in to detailed parts for study. It helps in identifying the problems and solutions.

4. Effective Synthesis

They are used as working models in the design of new program and systems. Elements of old approaches may be combined with new design ideas to give an effective processing plan.

5. Proper Program Documentation

They serve as a good program documentation; which is needed for the following reasons: Documented knowledge belongs to an organization and does not disappear with the departure of the programmer.

If programs are modified in future, flow chart helps the programmer in understanding the current logic.

This will aid greatly in future program conversion when new hardware/software packages are acquired.

6. Effective Coding

The programme flow charts acts as guide or blue print during the system analysis and programmer preparation phase. Instructions coded in programming languages may be checked against the flowchart to ensure that no stops are omitted.

7. Orderly Debugging and Testing of Programme

The flow chart helps in debugging process, i.e., it helps in detecting, locating and removing mistakes.

8. Effective Programme Maintenance

The charts help the programmer to concentrate attention on that part of the information flow which is to be modified.

Limitations of Flowcharts

1. **Complex logic :** When the program logic is complex the flow charts Quickly become complex and clumsy and lacks the clarity of decision making.
2. **Alterations and Modifications :** If alteration is required, the flow chart may require redrawing completely.

3. **Reproduction** : As the flowchart symbols cannot be typed reproduction of flowcharts is often a problem.
4. No uniform practice is followed for drawing flowcharts.
5. Link between conditions and actions, sometimes it is very difficult to establish link between various conditions and the actions to be taken there upon for a particular condition.
6. **Standardization** : Difficult to express procedure in English.
7. There are no obvious mechanism for progressing from one level of the design to the next.

Q28. What are different ways representing flow charts.

Ans :

Flow charts are composed of symbols of different sizes and shapes. These symbols have specialized meaning and functions in the flow charts. Flow charts are drawn using the basic symbols and functions.

Flow charts are used for two main purposes :

- To help the programmer develop the program logic.
- To serve as documentation for the completed program.

A flow chart is intended to focus on the program logic rather than on the specific task it has to be performed by the program. It reduces programming errors and maximizes the output. Flow chart uses certain symbols to represent the data.

Type of Flowcharts**1. System Flowcharts**

Expresses structure and relationship of a system in the form of flow chart. It describes the major operations, both manual and computer, data flow and the files used in , the processing and flow of processing control.

These are used by system analysts for analyzing and designing the system.

2. Modular Program Flow Charts

These flowcharts are used to represent specific function of a module under the application.

It represents the sequence of instructions that perform each procedure or the operation written in independent unit or module. They are also referred to as block diagrams.

3. Detail Program FlowCharts

These are the more comprehensive and elemental form of charts used by the programmers. Each symbol represents the executable steps and is suitable for coding as an instruction in the computer language.

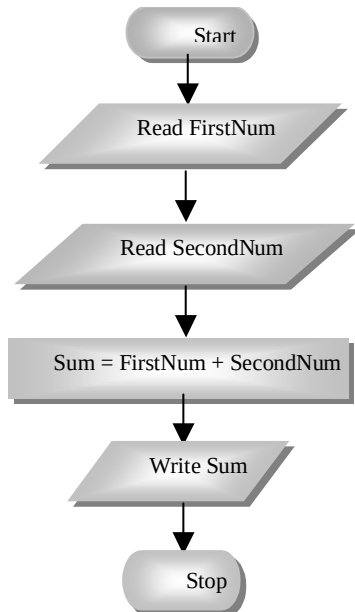
Constructing the flowcharts :

- Research the problem.
- Decide on the method of solution
- Plan the solution
- Outline the system by drawing the system flowchart.
- Check the system flowchart.
- Draw the detailed program flowchart for each program.
- Check the program flowchart.
- Test the program flowchart and system flowchart by using data that has been prepared.

Sample Flowcharts with Algorithms

1. Draw a flowchart for addition of two numbers.

Flowchart for addition of two numbers :

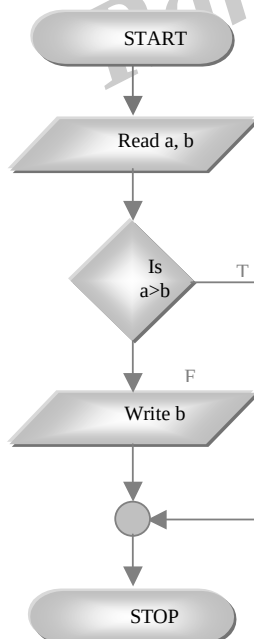


Algorithm for addition of two numbers:

Step 1: Start
 Step 2: Read FirstNumber
 Step 3: Read SecondNumber
 Step 4: Sum= FirstNumber + SecondNumber
 Step 5: Write (Sum)
 Step 6: Exit

Draw a flow chart to find large number between two numbers and write an algorithm for it.

Flowchart for finding large number between two numbers



Algorithm for finding large number between two numbers

Step 1: Start
 Step 2: Read **a** and **b**
 Step 3: IF **a > b** THEN Write (**a**)
 ELSE Write(**b**)
 Step 5: Stop

Q29. What is pseudocode ? Explain.

Ans : (Dec.-18, Dec.-17)

Pseudocode is made up of two words : pseudo and code. Pseudo means imitation and code refers to instructions, written in a programming language. As the name suggests, pseudocode is not a real programming code, but it models and may even look like programming code. Pseudocode is an outline of a program, written in a form that can be easily converted into real programming statements. Pseudocode uses plain English statements rather than symbols, to represent the processes of a computer program.

Pseudocode strikes a fine balance between the understandability and informality of a natural language like English, and the precision of a computer program code. It is somewhat halfway between English and a programming language. If an algorithm is written in English, the description may be at such a high level that it may prove difficult to analyze the algorithm and then to transform it into actual code. The goal of writing pseudocode is to provide a high-level description of an algorithm, which facilitates analysis and eventual coding, but at the same time suppresses many of the details that are insignificant.

Pseudocode uses some keywords to denote programming processes. Some of them are :

- **Input :** READ, OBTAIN, GET AND PROMPT
- **Output :** PRINT, DISPLAY, AND SHOW
- **Compute :** COMPUTE, CALCULATE, AND DETERMINE.
- **Initialize :** SET AND INITIALIZE
- **Add One :** INCREMENT

Since pseudocode is detailed yet readable, it can be inspected by the team of designers and programmers as a way to ensure that actual programming is likely to match design specifications. It is better to catch errors at the pseudocode stage rather than correcting them in later stages, as it would prove expensive. The benefit of pseudocode is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language.

Following are the general guidelines for developing pseudocodes :

1. Statements should be written in simple English and should be language independent. Pseudocodes describes only the logic plan to develop a program, it is not programming.
2. Steps must be understandable, and when the steps are followed, they must produce a solution to the specified problem. If the pseudocode is difficult for a person to read or translate into code, then something is wrong with the level of detail you have chosen to use.
3. Pseudocodes should be concise.
4. Each instruction should be written in a separate line and each statement in pseudocode should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.
5. Capitalise keywords such as READ, PRINT, and so on.
6. Each set of instructions is written from top to bottom, with only one entry and one exit.
7. It should allow for easy transition from design to implementation in programming language.

Benefits of Pseudocode

Programming can be complicated process when the program requirements are complex in nature. Pseudocode provides a simple method of developing the program logic as it uses everyday language to prepare a brief set of instruction in the order in which they appear in the completed program. It allows the programmer to focus on the steps required to solve a program rather than on how to use the computer language.

Following are the benefits of pseudocode :

1. Since pseudocode is language independent, it can be used by most programmers. It allows the developer to express the design in plain natural language.
2. It is easier to develop a program from a pseudocode than with a flowchart. Programmers do not have to think about syntaxes; they simply have to concentrate on the underlying logic. The focus is on the steps

to solve a problem rather than on how to use the computer language.

3. It is easy to translate pseudocode into a programming language. A step which can be accomplished by less experienced programmers.
4. The use of words and phrases in pseudocode, which are in line with basic computer operations, simplifies the translation from the pseudocode algorithm to a specific programming language.
5. Unlike flowcharts, pseudocode is compact and does not tend to run over many pages. Its simple structure and readability makes it easier to modify.
6. Pseudocodes allow programmers who work in different computer languages to talk to each other. They can be reviewed by groups more easily than real code.

Limitations of Pseudocode

Pseudocode is a very simple mechanism to simplify problem-solving logic, it has its limitations. Some of the most notable limitations are :

1. The main disadvantage of using pseudocode is that it does not provide visual representation of the program's logic.
2. There are no accepted standards for writing pseudocodes. Programmers use their own style of writing pseudocode.
3. Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step, an important one, in producing the final code.

1.13 STRATEGY FOR DESIGNING ALGORITHMS

Q30. Explain techniques for Designing Algorithms.

Ans :

Selecting a proper designing technique for a parallel algorithm is the most difficult and important task. Most of the parallel programming problems may have more than one solution. In this chapter, we will discuss the following designing techniques for parallel algorithms

- Divide and conquer
- Greedy Method
- Dynamic Programming
- Backtracking
- Branch & Bound
- Linear Programming

1. Divide and Conquer Method

In the divide and conquer approach, the problem is divided into several small sub-problems. Then the sub-problems are solved recursively and combined to get the solution of the original problem.

The divide and conquer approach involves the following steps at each level

Divide - The original problem is divided into sub-problems.

Conquer - The sub-problems are solved recursively.

Combine - The solutions of the sub-problems are combined together to get the solution of the original problem.

The divide and conquer approach is applied in the following algorithms -

- Binary search
- Quick sort
- Merge sort
- Integer multiplication
- Matrix inversion
- Matrix multiplication

2. Greedy Method

In greedy algorithm of optimizing solution, the best solution is chosen at any moment. A greedy algorithm is very easy to apply to complex problems. It decides which step will provide the most accurate solution in the next step.

This algorithm is called greedy because when the optimal solution to the smaller instance is provided, the algorithm does not consider the total program as a whole. Once a solution is considered, the greedy algorithm never considers the same solution again.

A greedy algorithm works recursively creating a group of objects from the smallest possible component parts. Recursion is a procedure to solve a problem in which the solution to a specific problem is dependent on the solution of the smaller instance of that problem.

3. Dynamic Programming

Dynamic programming is an optimization technique, which divides the problem into smaller sub-problems and after solving each sub-problem, dynamic programming combines all the solutions to get ultimate solution. Unlike divide and conquer method, dynamic programming reuses the solution to the sub-problems many times.

Recursive algorithm for Fibonacci Series is an example of dynamic programming.

Backtracking Algorithm

Backtracking is an optimization technique to solve combinational problems. It is applied to both programmatic and real-life problems. Eight queen problem, Sudoku puzzle and going through a maze are popular examples where backtracking algorithm is used.

In backtracking, we start with a possible solution, which satisfies all the required conditions. Then we move to the next level and if that level does not produce a satisfactory solution, we return one level back and start with a new option.

4. Branch and Bound

A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. It allows the algorithm to find parts of the solution space completely.

The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth-first search.

5. Linear Programming

Linear programming describes a wide class of optimization job where both the optimization criterion and the constraints are linear functions. It is a technique to get the best outcome like maximum profit, shortest path, or lowest cost.

In this programming, we have a set of variables and we have to assign absolute values to them to satisfy a set of linear equations and to maximize or minimize a given linear objective function.

1.14 STRUCTURED PROGRAMMING CONCEPT**Q31. Explain the concept of Structured Programming ?**

Ans :

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops.

It emerged in the late 1950s with the appearance of the ALGOL 58 and ALGOL 60 programming languages, with the latter including support for block structures. Contributing factors to its popularity and widespread acceptance, at first in academia and later among practitioners, include the discovery of what is now known as the structured program theorem in 1966, and the publication of the influential "Go To Statement Considered Harmful" open letter in 1968 by Dutch computer scientist Edsger W. Dijkstra, who coined the term "structured programming".

Structured Programming has been called a revolution in programming and one of the most important advances in computer software of the past two decades.

Both academic and industrial fields are inclined towards the philosophy and techniques of structured programming. Today, it is safe to say that virtually all software developers at least acknowledge the merits of the structured programming approach and use it in software development.

There is no standard definition of structured programming available but is often thought to be programming without the use of a GOTO statement. Indeed, structured programming does discourage the frequent use of GOTO but there is more to it than that.

With a large software package like the operating system and with business application programs which contain several thousands of lines of code (statements), it is a difficult task to keep track of all the details of the program at a time.

Structured programming tries to resolve this issue. But how does structured programming resolve this issue? It does so through the adoption of three ways (steps) which serve as the working definition of structured programming.

Definition of Structured Programming

It is the method of writing a computer program that uses :

- top-down analysis for program solving
- modularisation for program structure and organisation, and
- structured codes for the individual modules.

Organizations using structured programming have shown the following characteristics :

- **Programming Productivity** : Programmers write more program statements per day with fewer errors.
- **Program Economy** : The cost and time of program development and maintenance are reduced.
- **Program Simplicity** : Programs are easier to read, write, correct and maintain.

1.14.1 Top-Down Analysis

Q32. Explain about top Down approach with its process?

Ans :

Top-down and bottom-up are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories, and management and organization. In practice, they can be seen as a style of thinking, teaching, or leadership.

A top-down approach is essentially the breaking down of a system to gain insight into its compositional sub-systems in a reverse engineering fashion. In a top-down approach an overview of the system is formulated, specifying, but not detailing, any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements.

Top-down analysis simplifies or reduces the complexity of the progress of problem solving. It is not limited by the type of program. Top-down analysis is a general method for attacking any problem. It gives us a strategy to follow in solving all problems.

There are two essential ideas in top-down analysis :

- i) **Subdivision of a problem** means simply breaking a big problem into two or more smaller problems. Therefore, to solve the big problem, first these smaller problems have to be solved.
- ii) **Hierarchy of tasks** : The method of top-down analysis does not simply divide a problem into two or more smaller problems. It goes further than that. Each of these smaller problems is further subdivided. This process continues downward from one level to another level until no further break down is possible.

Four Basic Steps to Top-down Analysis

The activities for performing a top-down analysis of a problem can be summarised in four steps :

Step 1

Define the complete scope of a problem to determine the basic requirement for its solution.

Three factors must be considered in the definition of a programming problem : input, calculation and output.

- **Input** : What data will need to be processed by the program ?
- **Process** : What must be done with the input data? What type of processing is required ?
- **Output** : What information should the program produce? In what form should it be presented ?

Step 2

Based on the definition of the problem, divide it into two or more separate parts.

Step 3

Carefully define the scope of each of these separate tasks and subdivide them, if necessary, into two or more further smaller tasks.

Step 4

Repeat step 3. Every step at the lowest level describes a simple task which cannot be broken further.

1.14.2 Modular Programming**Q33. Explain about modular programming.**

Ans :

Modular programming is the process of subdividing a computer program into separate sub-programs.

A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system. Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.

Object-oriented programming (OOP) is compatible with the modular programming concept to a large extent. Modular programming enables multiple programmers to divide up the work and debug pieces of the program independently.

The benefits of using modular programming include :

1. Less code has to be written.
2. A single procedure can be developed for reuse, eliminating the need to retype the code many times.
3. Programs can be designed more easily because a small team deals with only a small part of the entire code.
4. Modular programming allows many programmers to collaborate on the same application.
5. The code is stored across multiple files.
6. Code is short, simple and easy to understand.
7. Errors can easily be identified, as they are localized to a subroutine or function.

8. The same code can be used in many applications.

9. The scoping of variables can easily be controlled.

Structured Code

After top-down analysis and modular structure the third and final aspect of structured programming involves the use of structured code. Structured programming is a method of coding (writing a program) that produces a well-organised module.

A high level language supports several control statements (also called structured control statements or structured code) to produce a well-organised (structured) module. These control statements represent conditional and repetitive type of executions. Each language has different syntax for these statements.

In PASCAL, the IF and CASE statements are examples of conditional execution whereas FOR, WHILE and REPEAT statements represent repetitive execution. In BASIC, FOR-NEXT and WHILE-WEND are examples of repetitive execution.

Let us consider GOTO statement of BASIC, which is a simple but not a structured control statement. The GOTO statement can bring control to any arbitrary point in a program which breaks normal control flow. A module which does not have a normal control flow is unorganised and unreadable.

The following example is a demonstration of a program using several GOTO statements. Please notice that at line numbers 20, 60 and 80, the normal control flow is broken.

For example from line number 60, control is going back to line number 40 instead of line number 70.

```
10 INPUT X
20 IF X < 0 THEN GOTO 90
30 G = X/2
40 R = X/G
50 G = (R + G)/2
60 IF ABS(R-G) > 0.0001 THEN GOTO 40
70 PRINT G
80 GOTO 100
```

```
90 PRINT "INVALID REPORT"
```

```
100 STOP
```

The structured version of this program using WHILE-WEND statement is written as :

```
INPUT X
```

```
IF X > 0
```

```
    THEN
```

```
    G = X/2
```

```
    R = X/G
```

```
        WHILE ABS(R + G) < 0.001
```

```
            R = X/G
```

```
            G = (R + G)/2
```

```
        WEND
```

```
        PRINT G
```

```
ELSE
```

```
    PRINT "INVALID INPUT"
```

```
END
```

Please notice that now there is no normal break of control flow. GOTOs is inevitable in unstructured languages but they can be and should be always avoided using structured programs except in some unavoidable situations.

1.14.3 Programming Process

Q34. Explain the steps for designing program?

Ans :

All programming involves creating something that solves a problem. The problems can range from something of great scientific or national importance, through to something as trivial as relieving personal boredom! Programming is complicated process, below is general overview of this 5 step procedure.

- 1 Clarify Programming Needs
- 2 Design the Program
- 3 Code the Program
- 4 Test the Program
- 5 Document and Maintain

1. Clarify Programming Needs

Knowing the objective is the first consideration. Is it a payroll or editing program? Knowing who the end user will be is also important. Determining the inputs and outputs is next. How will the program operate and what data is needed to make it happen. After this has been decided feasibility is the next consideration. How many programmers will it take, is the project within budget, does the project have a realistic outline. Finally, if the project is a go, then one must take measures to ensure the project is properly documented and analyzed.

Four mini steps are :

- Clarify desired processing
- Double - check feasibility of implementing the program
- Document the analysis
- kopp

2. Design the Program

Programs use algorithms which are like equations that tell the computer what task to perform. The aim of the programmer is to create algorithms that are clear and simple. Algorithms are expressed first in logical hierarchical form known as modularization. Using modules or (a complete thought) the programmer creates a logical thought process for the computer to follow. After that the program is broken down in greater detail using pseudocode. Pseudocode uses terms like if, else, and, then to relate the programs rules to the computer.

Two mini steps are :

- Determine program logic through top down approach and modularization, using a hierarchy chart
- Design details using pseudocode and/or flowcharts, preferably involving control structure.

3. Code the Program

After the program has been designed it must be coded or written. Using the pseudocode and logic requirements from step two an appropriate programming language must be selected. As stated in the introduction, coding languages differ in specifications and usability. Once the appropriate code language has been chosen, it is imperative that the programmer follow the syntax rules with as little deviation as possible in order for the program to have high accuracy.

Two mini steps are :

- Select the appropriate high-level programming language.
- Code the program in that language following the syntax carefully.

4. Test the Program

After the program is written it then enters the programming debugging and testing phase of the Program Development Life Cycle (PDLC). During this time the programmer will be looking for errors in both logic and syntax, as well as exploring other areas that may cause the program to either not work properly or to not run at all.

Debugging will uncover errors in both logic and syntax. Syntax errors will prevent the program from executing. They can be such simple things as misspelled words or can involve breaking the syntax rules of the programming language used.

On the other hand, logic errors will allow the program to run but will provide incorrect results. Errors of this kind may consist of merely using the wrong relational operator or other, larger, mistakes in writing formulas.

Once the programmer locates the errors they are then fixed and the program is run again. This will happen multiple times, often called “execute, check, and correct” (526), until the program runs flawlessly.

The program will then enter the testing phase.

5. Document and Maintain

Documentation should be ongoing from the very beginning because it is needed for those involved with program now and future. Upon completion User Documentation for commercial use, Operator Documentation for people who run computer systems, and Programmer Documentation for programmers charged with maintenance.

Four mini steps :

- Write user documentation
- Write operator documentation
- Write programmer documentation
- Maintain the program

1.15 OVERVIEW OF C

Q35. What is C-Programming language? Explain the features & importance?

Ans :

C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system.

Although C was designed for implementing system software, it is also widely used for developing portable application software.

C is one of the most popular programming languages. It is widely used on many different software platforms, and there are few computer architectures for which a C compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++ , which originally began as an extension to C.

C has facilities for structured programming and allows lexical variable scope and recursion, while a static type system prevents many unintended operations. In C, all executable code is contained within functions. Function parameters are always passed by value. Pass-by-reference is achieved in C by explicitly passing pointer values. Heterogeneous aggregate data types (struct) allow related data elements to be combined and manipulated as a unit. C program source text is free-format, using the semicolon as a statement terminator (not a delimiter).

C also exhibits the following more specific characteristics :

- non-nestable function definitions.
- variables may be hidden in nested blocks.
- partially weak typing; for instance, characters can be used as integers.
- low-level access to computer memory by converting machine addresses to typed pointers.
- function and data pointers supporting *ad hoc* run-time polymorphism.
- array indexing as a secondary notion, defined in terms of pointer arithmetic.

- a preprocessor for macro definition, source code file inclusion, and conditional compilation.
- complex functionality such as I/O, string manipulation, and mathematical functions consistently delegated to library routines.
- A relatively small set of reserved keywords.

Features of 'C' Language

1. **C is portable.** Code written on one machine can be easily moved to another. C provides the programmer with a standard library of functions that work the same on all machines. Also, C has a built-in preprocessor that helps the programmer isolate any system-dependent code.
2. **C is middle level language.** because it has both types of features i.e., High level as well as low level. High level language are very easy and fast to learn for readers and programs in this language are not machine dependent. Where as low level language programs are hardware dependent and it is very difficult to create and modify these because of the dependence on the machine.
3. **C is terse.** C has a very powerful set of operators, and some of these operators allow the programmer to access the machine at the bit level. The increment operator ++ has a direct analogue in machine language on many machines, making this an efficient operator. Indirection and address arithmetic can be combined within expressions to accomplish on one statement or expression what would require many statements in another language.

For many programmers this is both elegant and efficient. Software productivity studies show that programmers produce, on average, only a small amount of working code each day. A language that is concise explicitly magnifies the underlying productivity of the programmer.
4. **C is modular.** C supports one style of routine, the external function, for which arguments are passed *call-by-value*. The nesting of functions is not allowed. A limited

form of privacy is provided by using the storage class static within files. These features, along with tools provided by the operating system, readily support user-defined libraries of functions and modular programming.

5. **C is efficient on most machines** : Because certain constructs in the language are explicitly machine-dependent, C can be implemented in a manner that is natural with respect to the machine's architecture. Because a machine can do what comes naturally, compiled C code can be very efficient. Of course, the programmer must be aware of any code that is machine-dependent.
6. **C is appealing** : because of its powerful operators and its unfettered nature. A C programmer strives for functional modularity and effective minimalism. A C programmer welcomes experimentation and interaction.
7. C-language has an important facility called **extendability**. It means you can write your own file or functions subprogram according to the requirement.

Uses of C

C's primary use is for "system programming", including implementing operating systems and embedded system applications, due to a combination of desirable characteristics such as code portability and efficiency, ability to access specific hardware addresses, ability to "pun" types to match externally imposed data access requirements, and low runtime demand on system resources.

One consequence of C's wide acceptance and efficiency is that compilers, libraries, and interpreters of *other* programming languages are often implemented in C.

C is sometimes used as an intermediate language by implementations of other languages. This approach may be used for portability or convenience; by using C as an intermediate language, it is not necessary to develop machine-specific code generators.

C has also been widely used to implement end-user applications, but as applications became larger, much of that development shifted to other languages.

Importance of C

C language is a famous programming language due to its qualities. Some qualities are :

1. It is robust lang whose rich setup of built in functions and operator can be used to write any complex prog.
2. Programs written in C are efficient and fast due to several variety of data types and powerful operators.
3. The C compiler combines the capabilities of an assembly lang with the feature of high level language. Therefore it is well suited for writing both system software and business package.
4. C has only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.
5. C is portable language, this means that C programs written for one computer system can be run on another system, with little or no modification.
6. C language is well suited for structured programming, this requires user to think of a problems in terms of function or modules or block. A collection of these modules make a program debugging and testing easier.
7. C language has its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to the library with the availability of the large number of functions.

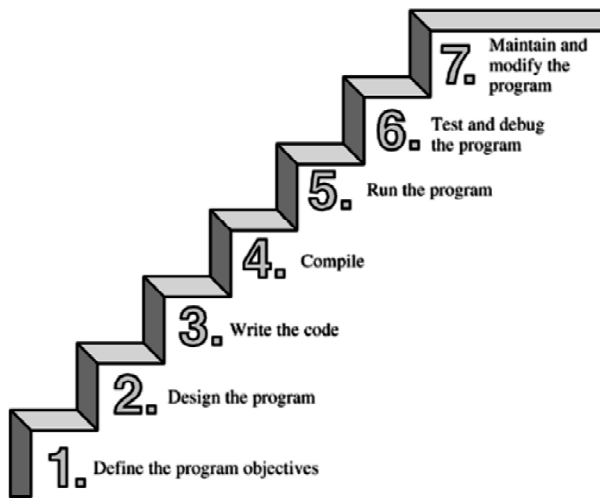
1.16 DEVELOPING PROGRAMS IN C

Q36. Explain the steps used to develop the program?

Ans :

C, as you've seen, is a compiled language. If you are accustomed to using a compiled language, such as Pascal or FORTRAN, you will be familiar with the basic steps in putting together a C program. However, if your background is in an interpreted language, such as BASIC, or in a graphical

interface-oriented language, such as Visual Basic, or if you have no background at all, you need to learn how to compile. We'll look at that process soon, and you'll see that it is straightforward and sensible. First, to give you an overview of programming, let's break down the act of writing a C program into seven steps (see Figure 1.3). Note that this is an idealization. In practice, particularly for larger projects, you would go back and forth, using what you learned at a later step to refine an earlier step.



The seven steps of programming.

Step 1: Define the Program Objectives

Naturally enough, you should start with a clear idea of what you want the program to do. Think in terms of the information your program needs, the feats of calculation and manipulation the program needs to do, and the information the program should report back to you. At this level of planning, you should be thinking in general terms, not in terms of some specific computer language.

Step 2: Design the Program

After you have a conceptual picture of what your program ought to do, you should decide how the program will go about it. What should the user interface be like? How should the program be organized? Who will the target user be? How much time do you have to complete the program?

You also need to decide how to represent the data in the program and, possibly, in auxiliary files, as well as which methods to use to process the data. When you first learn programming in C, the choices

will be simple, but as you deal with more complex situations, you'll find that these decisions require more thought. Choosing a good way to represent the information can often make designing the program and processing the data much easier.

Again, you should be thinking in general terms, not about specific code, but some of your decisions may be based on general characteristics of the language. For example, a C programmer has more options in data representation than, say, a Pascal programmer.

Step 3: Write the Code

Now that you have a clear design for your program, you can begin to implement it by writing the code. That is, you translate your program design into the C language. Here is where you really have to put your knowledge of C to work. You can sketch your ideas on paper, but eventually you have to get your code into the computer. The mechanics of this process depend on your programming environment. We'll present the details for some common environments soon. In general, you use a text editor to create what is called a source code file. This file contains the C rendition of your program design. Listing 1.1 shows an example of C source code.

Example of C Source Code

```
#include <stdio.h>

int main(void)
{
    int dogs;

    printf("How many dogs do you have?\n");
    scanf("%d", &dogs);
    printf("So you have %d dog(s)\n", dogs);
    return 0;
}
```

As part of this step, you should document your work. The simplest way is to use C's comment facility to incorporate explanations into your source code. Chapter 2, "Introducing C," will explain more about using comments in your code.

Step 4: Compile

The next step is to compile the source code. Again, the details depend on your programming

environment, and we'll look at some common environments shortly. For now, let's start with a more conceptual view of what happens.

Recall that the compiler is a program whose job is to convert source code into executable code. Executable code is code in the native language, or machine language, of your computer. This language consists of detailed instructions expressed in a numeric code. As you read earlier, different computers have different machine languages, and a C compiler translates C into a particular machine language. C compilers also incorporate code from C libraries into the final program; the libraries contain a fund of standard routines, such as `printf()` and `scanf()`, for your use. (More accurately, a program called a linker brings in the library routines, but the compiler runs the linker for you on most systems.) The end result is an executable file containing code that the computer understands and that you can run.

The compiler also checks that your program is valid C. If the compiler finds errors, it reports them to you and doesn't produce an executable file. Understanding a particular compiler's complaints is another skill you will pick up.

Step 5: Run the Program

Traditionally, the executable file is a program you can run. To run the program in many common environments, including MS-DOS, Unix, Linux consoles, just type the name of the executable file. Other environments, such as VMS on a VAX, might require a run command or some other mechanism. Integrated development environments (IDEs), such as those provided for Windows and Macintosh environments, allow you to edit and execute your C program from within the IDE by selecting choices from a menu or by pressing special keys. The resulting program also can be run directly from the operating system by clicking or double-clicking the filename or icon.

Step 6: Test and Debug the Program

The fact that your program runs is a good sign, but it's possible that it could run incorrectly. Consequently, you should check to see that your

program does what it is supposed to do. You'll find that some of your programs have mistakes—bugs, in computer jargon. Debugging is the process of finding and fixing program errors. Making mistakes is a natural part of learning. It seems inherent to programming, so when you combine learning and programming, you had best prepare yourself to be reminded often of your fallibility. As you become a more powerful and subtle programmer, your errors, too, will become more powerful and subtle.

You have many opportunities to err. You can make a basic design error. You can implement good ideas incorrectly. You can overlook unexpected input that messes up your program. You can use C incorrectly. You can make typing errors. You can put parentheses in the wrong place, and so on. You'll find your own items to add to this list.

Fortunately, the situation isn't hopeless, although there might be times when you think it is. The compiler catches many kinds of errors, and there are things you can do to help yourself track down the ones that the compiler doesn't catch. This book will give you debugging advice as you go along.

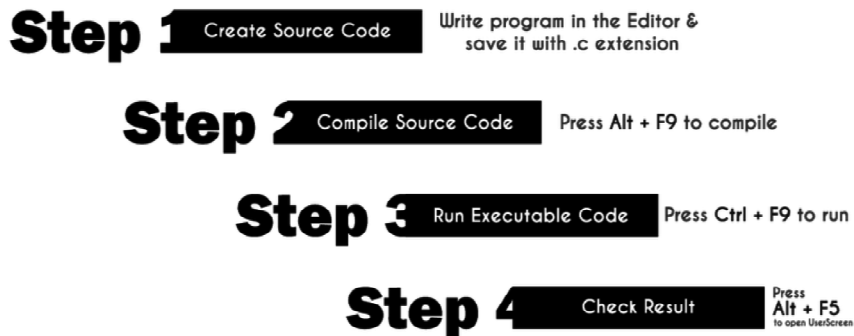
Step 7: Maintain and Modify the Program

When you create a program for yourself or for someone else, that program could see extensive use. If it does, you'll probably find reasons to make changes in it. Perhaps there is a minor bug that shows up only when someone enters a name beginning with Zz, or you might think of a better way to do something in the program. You could add a clever new feature. You might adapt the program so that it runs on a different computer system. All these tasks are greatly simplified if you document the program clearly and if you follow sound design practices.

Q37. Explain Process of creating compiling and executing c program.

Ans :

To create and execute C programs in Windows Operating System, we need to install Turbo C software. We use the following steps to create and execute C programs in Windows OS...



Step 1: Creating Source Code

Source code is a file with C programming instructions in high level language. To create source code, we use any text editor to write the program instructions. The instructions written in the source code must follow the C programming language rules. The following steps are used to create source code file in Windows OS...

- Click on Start button
- Select Run
- Type cmd and press Enter
- Type cd c:\TC\bin in the command prompt and press Enter
- Type TC press Enter
- Click on File -> New in C Editor window
- Type the program
- Save it as FileName.c (Use shortcut key F2 to save)

Step 2: Compile Source Code (Alt + F9)

Compilation is the process of converting high level language instructions into low level language instructions. We use the shortcut key Alt + F9 to compile a C program in Turbo C.

Whenever we press Alt + F9, the source file is going to be submitted to the Compiler. On receiving a source file, the compiler first checks for the Errors. If there are any Errors then compiler returns List of Errors, if there are no errors then the source code is converted into object code and stores it as file with .obj extension. Then the object code is given to the Linker. The Linker combines both the object code and specified header file code and generates an Executable file with .exe extension.

Step 3: Executing / Running Executable File (Ctrl + F9)

After completing compilation successfully, an executable file is created with .exe extension. The processor can understand this .exe file content so that it can perform the task specified in the source file.

We use a shortcut key Ctrl + F9 to run a C program. Whenever we press Ctrl + F9, the .exe file is submitted to the CPU. On receiving .exe file, CPU performs the task according to the instruction written in the file. The result generated from the execution is placed in a window called User Screen.

Step 4: Check Result (Alt + F5)

After running the program, the result is placed into User Screen. Just we need to open the User Screen to check the result of the program execution. We use the shortcut key Alt + F5 to open the User Screen and check the result.

The file which contains c program instructions in high level language is said to be source code. Every c program source file is saved with .c extension, for example Sample.c.

Whenever we press Alt + F9 the source file is submitted to the compiler. Compiler checks for the errors, if there are any errors, it returns list of errors, otherwise generates object code in a file with name Sample.obj and submit it to the linker. Linker combines the code from specified header file into object file and generates executable file as Sample.exe. With this compilation process completes.

Now, we need to Run the executable file (Sample.exe). To run a program we press Ctrl + F9. When we press Ctrl + F9 the executable file is submitted to the CPU. Then CPU performs the task according to the instructions written in that program and place the result into UserScreen.

Then we press Alt + F5 to open UserScreen and check the result of the program.

Important Points

C program file (Source file) must save with .c extension.

Compiler converts complete program at a time from high level language to low level language.

Input to the compiler is .c file and output from the compiler is .exe file, but it also generates .obj file in this process.

Compiler converts the file only if there are no errors in the source code.

CPU places the result in User Screen window.

Overall Process

- Type the program in C editor and save with .c extension (Press F2 to save).
- Press Alt + F9 to compile the program.
- If there are errors, correct the errors and recompile the program.
- If there are no errors, then press Ctrl + F9 to execute / run the program.
- Press Alt + F5 to open User Screen and check the result.

1.17 PARTS OF SIMPLE C PROGRAM**Q38. Explain different parts of C Program with example ?**

Ans :

Before we start executing programs in C Turbo C/C++ Compiler we must know some basic parts of a C program.

- **#include** – The #include is known as a preprocessor directive and is used to tell the C preprocessor to find the stdio file with extension .h. <stdio.h> stand for standard input output stream header file and contains information for printf, scanf etc.

The file stdio.h will literally be included (copied) into your source code right at the place that the #include directive occurs. The stdio.h file includes numerous definitions of functions that are part of the C standard library for doing input/output. It includes prototypes for all of these functions.

- **Prototypes** – A function prototype defines the function signature for a function. As shown in the figure, the function prototype for the myFunction function lists:
 - What is returned from the function
 - The name of the function
 - The types of each parameter to the function

A function prototype is used to define the signature of the function so that later calls to the function know what that signature is.

Prototypes are required in a C program like this. Note that you CAN swap the function definitions in the program above, and then get rid of the prototype.

- **main()** – Execution of a program starts from a main() function. It defines the point from where the execution of the program starts. Anything written between opening curly brace and ending curly brace of main is executed.
- **printf()** – This is the standard way of producing output. The functionality of printf() is referenced in stdio.h by the C compiler, thus it always work in the same way.
- **scanf()** – This is the standard way of taking input from user. The functionality of scanf() is also referenced in stdio.h by the C compiler.
- **comments** – Comments are information given by the program to make a program readable and easy to understand. It reduces the complexity of a program. Anything written as comments is ignored by the compiler.

There are two ways of writing comments :

1) Single line comment

Syntax :

```
//Your comments here
```

Example :

```
// Hello this is my first C program
```

2) Multiline comment

Syntax:

```
/* Your comment
```

```
Your comment*/
```

Example :

```
/* This is my first C program
```

```
and i am very excited about it */
```

Note : It should be noted that some compiler does not include header file "conio.h". It is also not included in compiler of Linux or any other Unix based operating system. So in case using it displays an error remove this and the function associated with it i.e getch().


```

#include<stdio.h>    //This tells the compiler about the input/output functions such
                    as printf(), scanf()
#include<conio.h>    //It is used for getch() function
int main()          // It is the entry point of a program
{                  // Program begins with this curly braces
    printf("Congratulation you successfully run your first program"); // to print in the
                                                                    output screen

    getch();        // to hold the output screen
    return 0;       // tell the OS that the program exited without error
}                  // Program end with this curly braces

```

Output :

Congratulation you successfully run your first program

When we execute this program (press Ctrl+F9 to execute in Turbo C or F9 in Dev C++). If you want to execute this program in Dev C++ then there is a minor change. Replace void main() with int main() as Dev C++ compiler does not support void as return type for main(). Also replace getch() with return 0.

return 0 means that the program is terminated successfully and the compiler is returning back the control to the computer

Now on execution the program will display "Congratulation you successfully run your first program" (without double quotes) as output on monitor. Any string you pass within double quotes through printf is sent to console output (monitor) i.e it is displayed as it is on the monitor.

1.18 STRUCTURE OF A C PROGRAM

Q39. Explain the structure of C Program?

Ans :

(Nov.-20, June-19, Dec.-18, Dec.-17, June-17, Imp.)

C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in figure below.

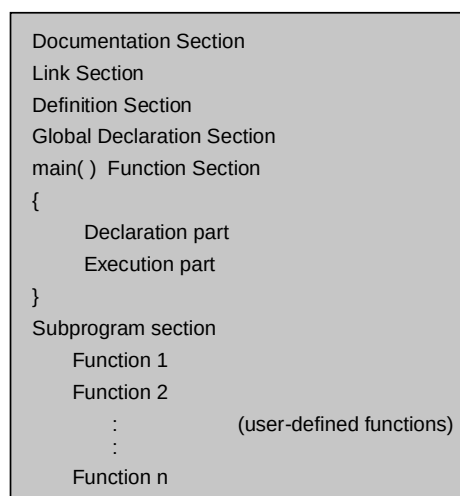


Fig.: An overview of a C program

- (i) **Document Section :** This section consists of a set of comment lines giving the name of the program, which the programmer would like to use later.

```
#include <stdio.h> ← Preprocessor director
#define VALUE 10 ← Preprocessor director

int global_var;      ← Global variable declaration
void main(void)
{
    /* This is the beginning of the program */      ← Comments
    int local_var; ← Local variable declaration
    local_var = 5; ← Variable definition
    global_var = local_var + VALUE;
    printf("Total sum is: %d\n", global_var); /*Printout the result */
}
```

- (ii) **Link Section :** The link section provides instructions to the compiler to link functions from the system library. Link section lists files to be included and preprocessed before compiling the program. These included files provide instructions to the compiler to link functions from the system library

For example :

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

- (iii) **Definition Section :** The definition section defines all symbolic constants.

```
#define PI 3.14
```

Here PI is the symbolic constant

- (iv) **Global Declaration Section :** Here global variables are declared. Global 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.

- (v) **main() Function Section :** Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. These two parts must be in between opening and closing braces.

- The declaration part declares all the variables and functions, which are to be used in the executable part.
- The executable part contains all the instructions to be executed. The program execution begins at the opening brace and ends at the closing brace. All statement in the declaration and executable parts end with a semicolon.

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

All the section, except main function may be absent when they are not required. Main is the unique section of the program i.e. use of more than one main() is not allowed.

Q40. What is C - Preprocessor?

Ans :

The **C preprocessor (cpp)** is the preprocessor for the C programming language. In many C implementations, it is a separate program invoked by the compiler as the first part of translation. The preprocessor handles directives for source file inclusion (`#include`), macro definitions (`#define`), and conditional inclusion (`#if`). The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other types of files.

Most C preprocessor features are inactive unless you give specific commands to request their use. But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions.

Most preprocessor features are active only if you use preprocessor commands to request their use.

Preprocessor commands are lines in your program that start with `#`. The `#` is followed by an identifier that is the *command name*. For example, `#define` is the command that defines a macro. Whitespace is also allowed before and after the `#`.

The set of valid command names is fixed. Programs cannot define new preprocessor commands. Some command names require arguments; these make up the rest of the command line and must be separated from the command name by whitespace. For example, `#define` must be followed by a macro name and the intended expansion of the macro.

Header Files

A header file is a file containing C declarations and macro definitions to be shared between several source files. You request the use of a header file in your program with the C preprocessor command `#include`.

Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place.

#include

Both user and system header files are included using the preprocessor command `#include`. It has three variants :

#include <file>

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `-I`. The option `-nostdinc` inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of `#include` is slightly special because comments are not recognized within the `<...>`. Thus, in `#include <x/*y>` the `/*` does not start a comment and

the command specifies inclusion of a system header file named 'x/y'. A header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument *file* may not contain a '>' character. It may, however, contain a '<' character.

#include "file"

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to.

#include anything else

This variant is called a *computed #include*. Any '#include' directive whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded (see section Macros). When this is done, the result must fit one of the above two variants—in particular, the expanded text must in the end be surrounded by either quotes or angle braces. This feature allows you to define a macro which controls the file name to be used at a later point in the program.

The '#include' command works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '#include' command. For example, given two files as follows :

```
/* File program.c */
int x;
#include "header.h"
main( )
{
    printf (test( ));
}
```

```
/* File header.h */
char *test( );
```

the output generated by the C preprocessor for 'program.c' as input would be

```
int x;
char *test( );
main( )
{
    printf (test( ));
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file.

#define

If we write

```
#define [identifier name] [value]
```

whenever [identifier name] shows up in the file, it will be replaced by [value].

If you are defining a constant in terms of a mathematical expression, it is wise to surround the entire value in parentheses:

```
#define PI_PLUS_ONE (3.14 + 1)
```

By doing so, you avoid the possibility that an order of operations issue will destroy the meaning of your constant :

```
x = PI_PLUS_ONE * 5;
```

Without parentheses, the above would be converted to

```
x = 3.14 + 1 * 5;
```

which would result in 1 * 5 being evaluated before the addition, not after.

It is also possible to write simply

```
#define [identifier name]
```

which defines [identifier name] without giving it a value. This can be useful in conjunction with another set of directives that allow conditional compilation.

Q41. Define Character Set?*Ans :*

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most personal, micro, mini and mainframe computers.

The characters in C are grouped into the following categories:

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

The compiler ignores white spaces unless they are a part of string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

C Character Set*Letters*

Uppercase A.....Z

Digits

All decimal digits 0 9

White Spaces

Blank space

Horizontal tab

Carriage return

New line

Form feed

Special Characters

, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
: colon	- minus sign
? question mark	+ plus sign
' apostrophe	< opening angle bracket
" quotation mark	(or less than sign)
! exclamation mark	> closing angle bracket
vertical bar	(or greater than sign)
/ slash	(left parenthesis
\ backslash	) right parenthesis
~ tilde	[left bracket
_ under score	] right bracket
\$ dollar sign	{ left brace
% percent sign	} right brace
	# number sign

1.19 COMMENTS

Q42. Explain how comments are used in programming language?

Ans :

(Dec.-16)

Comments can be put just about anywhere you want in your program. As a general rule, you can put a comment wherever you can put a space, since comments are simply replaced by a single space in the program.

A *comment* begins with `/*` and continues until a corresponding `*/` is encountered. Notice that comments can continue over multiple lines.

You can include comments within your functions and even within a particular statement. About the only place you cannot include a comment is in the middle of a word (such as a function or variable name, or a reserved word) in the program.

When you have “comments out” code – that is, make the code initiative by putting it within the `/*` and `*/` markers. For example, consider the following variant of the first example :

```
/* program with awkward comments. */
main ( )
{
    printf ("It's traditional to say hello");
    printf ("in your first C program.\n"); /* watch
    printf ("So, hello there.\a");      this */
}
```

If you compile and run this program, you'll find that only one line is written. This is because the third call to `printf ( )` has been commented out. The compiler begins with the `/*` and ignore all input read until the `*/`. In the example, this includes the instruction to write the third string.

1.20 PROGRAM STATEMENTS

Q43. What are programming statements?

Ans :

A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements. All C program statements are terminated with a semicolon (`;`). Different types of program statement available in C is shown in Figure below.

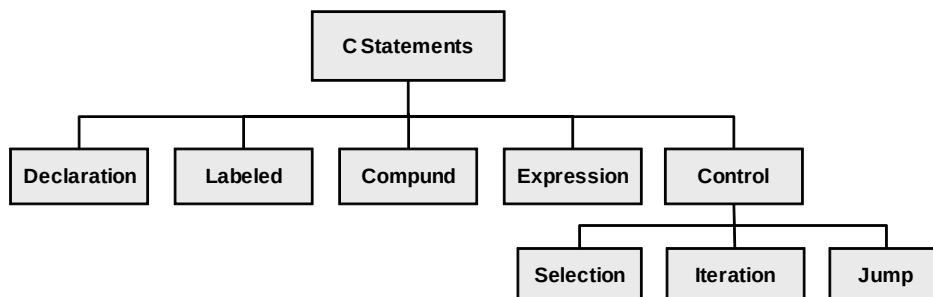


Fig.: C Statements

Declaration

The declaration tells the computer which storage locations or variables to use in the program. The declaration statement specifies type of the information (or data) to be stored, in turn memory space required for that information and a label (or name) to that memory location to access it in the further program. In C, the simple declaration statement looks like,

```
int rate ;
```

where **rate** is the variable (or name) and **int** is the datatype. The compiler decides how much memory space is required for this information and label that space with **rate**.

Labeled Statements

A statement can be preceded by a label. Three types of labels exist in C.

A simple identifier followed by a colon (:) is a label. Usually, this label is the target of a **goto** statement. Within **switch** statements, **case** and **default** labeled statements exist. A statement of the form

```
case constant-expression : statement
```

indicates that control will pass to this statement if the value of the control expression of the **switch** statement matches the value of the constant-expression. A statement of the form

```
default : statement
```

indicates that control will pass to this statement if the control expression of the **switch** statement does not match any of the constant-expressions within the **switch** statement. If the **default** statement is omitted, the control will pass to the statement following the **switch** statement.

Compound Statements

A compound statement is the way C groups multiple statements into a single statement. It consists of multiple statements and declarations within braces (i.e. { and }). In the ANSI C Standard of 1989-1990, a compound statement contained an optional list of declarations followed by an optional list of statements; in more recent revisions of the Standard, declarations and statements can be freely interwoven through the code. The body of a function is also a compound statement by rule.

Expression Statements

An expression statement consists of an optional expression followed by a semicolon (;). If the expression is present, the statement may have a value. If no expression is present, the statement is often called the null statement.

The **printf** function calls are expressions, so the statements above are expression statements.

Control Statements

Control statements enable us to specify the flow of program control; i.e., the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another. There are four types of control statements in C :

- 1) **Selection statements** : This type of statements allow a program to select a particular execution path from a set of one or more alternatives. Various forms of the *if..else* statement belong to this category.

- 2) **Iteration statements** : *Iteration statements* are used to execute a group of one or more statements repeatedly. *while*, *for*, and *do..while* statements falls under this group.
- 3) **Jump statements** : Jump statements cause an unconditional jump to another statement elsewhere in the code. They are used primarily to interrupt switch statements and loops. There are basically 3 jumping statements:
- break jumping statements.
 - continue jumping statements.
 - goto jumping statements.

1.21 C TOKENS

Q44. What are C - Tokens?

Ans :

(Dec.-18, Dec.-17)

In a C program the smallest individual units are known as C tokens. A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. Tokens are basic building blocks of C Programming.

C has six types of tokens as shown in fig.below. C programs are written using these tokens are the syntax of the language.

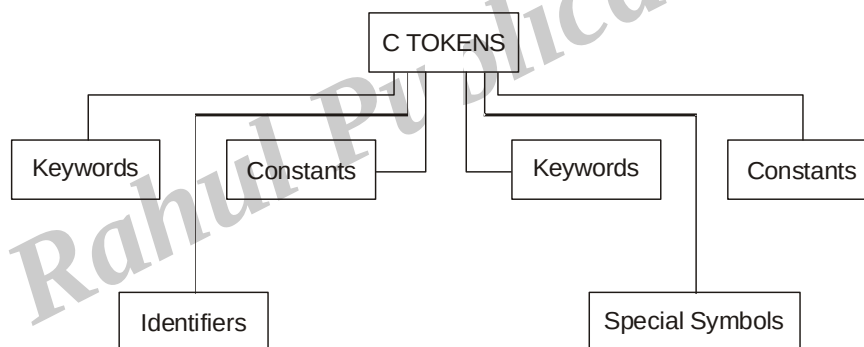


Fig.: C tokens and examples

Basic Building Blocks and Definition

Token	Meaning
Keyword	A variable is a meaningful name of data storage location in computer memory. When using a variable you refer to memory address of computer
Constant	Constants are expressions with a fixed value
Identifier	The term identifier is usually used for variable names
String	Sequence of characters
Special Symbol	Symbols other than the Alphabets and Digits and white-spaces
Operators	A symbol that represent a specific mathematical or non mathematical action

1.22 KEYWORDS AND IDENTIFIERS**Q45. What are Keywords & Identifiers in C?***Ans :***(Dec.-17)**

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. The list of all keywords of C are listed in Table below. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

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. Identifier names must be unique. They are created to give unique name to a C entity to identify it during the execution of a program. For example :

```
int money;
```

```
double accountBalance;
```

Here, **money** and **accountBalance** are identifiers.

Also remember, identifier names must be different from keywords. You cannot use **int** as an identifier because **int** is a keyword.

Rules for an Identifier

- An Identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
- The first character of an identifier can only contain alphabet(a-z , A-Z) or underscore (_).
- Identifiers are also case sensitive in C. For example name and Name are two different identifier in C.
- Keywords are not allowed to be used as Identifiers.
- No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

Identifier vs Keyword

Identifier	Keyword
Identifiers are names that are given to various program elements, such as variable, function and arrays.	C take some reserved word called keyword, they have predefine meaning in C.
Identifiers consist of letters and digits.	Keyword consist only letter.
Identifier's first character must be a letter.	Keyword's all character is letter.
Identifiers Upper & lowercase letter is use.	Keywords are all lowercase.
Upper and lowercase are not equivalent.	Upper and lowercase are also not equivalent.
Like: X, sum_5, _weather etc. But 4th is not identifier cause identifier first character must a letter.	Like: auto, short, long etc.

1.23 DATA TYPES**Q46. What do you mean by datatype ? How many data types does C - Language consist ?****Ans :****(Dec.-19, June-18, Dec.-17)**

C language provides rich set of is rich in its *data types*. C has a concept of '*data types*' which are used to define a variable before its use.

A C language programmer has to tell the system before-hand, the type of numbers or characters he is using in his program. These are data types. There are many data types in C language. A C programmer has to use appropriate data type as per his requirement.

C data types are used to :

- Identity the type of a variable when it declared.
- Identify the type of the return value of a function.
- Identify the type of a parameter expected by a function.

ANSI C supports 3 different data types

1. Primary (or fundamental) data types
2. User-defined data types
3. Derived data types

All C compilers support four fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), and double-precision floating point (**double**). Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Figure. The size and range of the basic data types are given in Table below.

1. Primary Data Types

These are fundamental data types in C namely integer(int), floating(float), character(char) and void.

Integral Type			
Integer		Character	
singed type int short int long int	unsigned type unsigned int unsigned short int unsigned long int	char signed char unsigned char	
Floating Point Type			
float	double	long double	void
Data types	Range of values		
char	-128 to 127		
int	-32,768 to 32,767		
float	3.4e-38 to 3.4e+38		
double	1.7e-308 to 1.7e+308		

(i) 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 (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (i.e., -2^{15} to $+2^{15} - 1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

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

For example, **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.

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration is a signed number. Table shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

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	8	–128 to 127
signed short int		
unsigned short int	8	0 to 255
long int or	32	–2,147,483,648 to 2,147,483,647
signed long int		
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E–38 to 3.4E+38
double	64	1.7E–308 to 1.7E+308
long double	80	3.4E–4932 to 1.1E+4932

(ii) Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), 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* numbers.

Remember that **double** type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits.

(iii) Void Type

Using void data type, we can specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

(iv) 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**. While **unsigned chars** have values between 0 and 255, **signed chars** have values from –128 to 127.

2. User-Defined Data Types

User-Defined Type Declaration : C supports a feature known as “type definition” that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form :

```
typedef type identifier;
```

When *type* refers to an existing data type and “identifier” refers to the “new” name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is ‘new’ only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are :

```
typedef int score;
typedef float credits;
```

Here **units** symbolizes **int** and **marks** symbolizes **float**. They can be later used to declare variables as follows :

```
score    social,
computer ;
credits  Fyear[5]  Syear[5];
```

social and computer are declared as **int** variable and Fyear[5] and Syear[5] are declared as 5 element floating point array variables.

The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type is enumerated data type provided by ANSI standard. It is defined as follows:

```
enum identifier {value 1, value 2, ...
valuen};
```

The “identifier” is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants*). After this definition, we can declare variables to be of this ‘new’ type as below:

```
enum identifier v1, v2, ... vn;
```

The enumerated variables v1, v2, ... vn can only have one of the values *value 1, value 2, ..., valuen*.

The assignments of the following types are valid:

```
v1 = value 3;
v5 = value 1;
```

An example : **enum day** {Monday, Tuesday, ..., Sunday};

```
enum day week_st,
week_end;

week_st = Monday;
week_end = Friday;
if(week_st = Tuesday)
week_end = Saturday;
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant *value 1* is assigned 0, *value 2* is assigned, 1, and so on. However the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example :

```
enum day {Monday = 1, Tuesday, ...
Sunday};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

The definition and declaration of enumerated variables can be combined in one statement.

Example : **enum day** {Monday, ...Sunday} week_st, week_end;

3. Derived Data Types

Derived data types are derived from the standard data type by adding some additional relationship with the various elements of the primary or standard data types. Note that derived data type may be used for representing a single value or multiple values. As these data types have different structure depending on the C-coding, these are also called Structured data type. These are also called secondary data type.

These are further sub divided into three categories :

- (a) Arrays and Strings
- (b) Structures
- (c) Unions

1.24 VARIABLES

Q47. What are variables ? How do you initialize the variable ?

Ans :

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.

Use of variables in C

The main purpose of variables is to store data in memory for later use, and that data values can be change.

Rules on choosing variable names :

As mentioned earlier, variable names may consist of letters, digits, and the underscore (_) character, subject to the following conditions :

- 1) Variable name can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.
- 2) Special characters like #, \$ are not allowed.
- 3) They must begin with a letter. Some systems permit underscore as the first character.
- 4) ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.
- 5) Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL**.
- 6) C keywords cannot be used as variable names.
- 7) White space is not allowed.
- 8) Variables names are case sensitive.

Variable Definition in C

Syntax :

```
type variable_name;  
or  
type variable_name, variable_name, variable_name;
```

Variable Definition and Initialization

Example :

```
int    width, height=5;  
char   letter='A' ;  
float   age, area;  
double d;  
/* actual initialization */  
width = 10;  
age = 26.5;
```

Variable Assignment

Variable assignment is a process of assigning a value to a variable.

Example :

```
int width = 60;  
int age = 31;
```

1.25 CONSTANTS

Q48. What are constants ? Explain its types?

Ans :

(Dec.-19, Dec.-17)

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

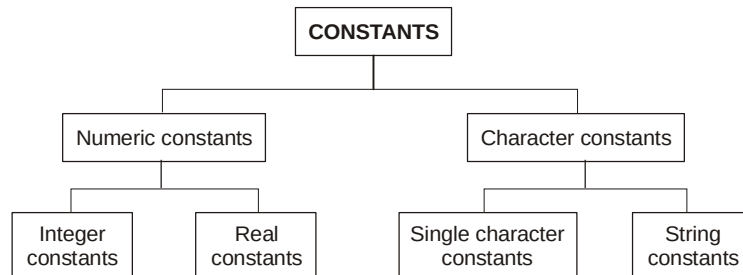


Fig.: Basic types of C constants

Integer Constants

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

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are :

123 – 241 0 764321 +68

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

25 950 50,000 \$2000

are illegal numbers. Note that ANSI C supports *unary plus* which was not defined earlier.

An *octal integer* constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are :

027 0 0435 0551

Example :

Representation of integer constants on a 16-bit computer.

The program in below illustrates the use of integer constants on a 16-bit machine. The output in Figure below shows that the integer values large than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```

main ( )
{
    printf("Integer values\n\n");
    printf("%d, %d %d\n", 32767, 32767+1, 32767+10);
    printf("\n");
    printf("Long integer values\n\n");
    printf("%1d %1d %1d\n", 32767L, 32767L+1L, 32767L+10L);
}
  
```

Output :

Integer values
32767 -32768 -32759
Long integer values
32767 32768 32777

Fig.: Representation of integer constants on 16-bit machine

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distance heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are :

0.0083 - 0.75 435.36 + 247.0

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 digits after the decimal point. That is,

215. .95 -.71 +.5

are all valid numbers.

A real number may also be expressed in *exponential* (or *scientific*) notation. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10². The general form is :

mantissa e exponent

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus sign*. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent cause the decimal point to "float", this notation is said to represent a real number in *floating point form*. Examples of legal floating-point constants are :

0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quota marks.

Example of character constants are :

'5' 'X' ';' ''

Note that the character constant '5' is not the same as the *number* 5. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

printf("%d", 'a');

would print the number 97, the ASCII value of the letter a. Similarly, the statement

printf("%c", '9');

would output the letter 'a'.

String Constant

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

Text enclosed in double quote characters (such as "example") is a *string constant*. It produces a block of storage whose type is *array of char*, and whose value is the sequence of characters between the double quotes, with a null character (ASCII code 0) automatically added at the end. All the escape sequences for character constants work here too.

The compiler merges a series of adjacent string constants into a single string constant (before automatically adding a null character at the end). For example, "sample" "text" produces the same single block of storage as "sampletext".

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol 'n' stands for newline character. A list of such backslash character constants is given in table below.

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\"'	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

Table : Backslash Character Constants

1.25.1 Symbolic Constants

Q49. What are symbolic constants? Explain with example?

Ans :

A *symbolic constant* is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character or a string constant. Thus, a symbolic constant allows a name to appear in place of a numeric constant, a character constant or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

Symbolic constants are usually defined at the beginning of a program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc. that the symbolic constants represent.

A symbolic constant is defined by writing

```
# define name text
```

where *names* represents a symbolic name, typically written in uppercase letters, and *text* represents the sequence of characters that is associated with the symbolic name. Note that *text* does not end with a semicolon, since a symbolic constant definition is not a true C statement. Moreover, if *text* were to end with a semicolon, this semicolon would be

treated as though it were a part of the numeric constant, character constant or string constant that is substituted for the symbolic *name*.

Example :

A C program contains the following symbolic constant definitions.

```
#define TAXRATE 0.23
#define PI 3.141593
#define TRUE 1
#define FALSE 0
#define FRIEND "Susan"
```

Notice that the symbolic names are written in uppercase, to distinguish them from ordinary C identifiers. Also, note that the definitions do not end with semicolons.

Now suppose that the program contains the statement.

```
area = PI * radius * radius;
```

During the compilation process, each occurrence of a symbolic constant will be replaced by its corresponding text. Thus, the above statement will become

```
area = 3.141593 * radius * radius ;
```

Now suppose that a semicolon had been (incorrectly) included in the definition for PI, i.e.,

```
#define PI 3. 141593;
```

The assignment statement for area would then become

```
area = 3.141539 ; * radius * radius ;
```

Note the semicolon preceding the first asterisk. This is clearly incorrect, and it will cause an error in the compilation.

The substitution of text for a symbolic constant will be carried out anywhere beyond the #define statement, *except* within a string. Thus, any text enclosed by (double) quotation marks will be unaffected by this substitution process.

1.26 OPERATORS

Q50. Define operator in C ? Explain all types of operators available in C with example?

Ans : (Dec.-18, June-18, Dec.-16, Imp.)

'C' has many built-in operators. An operator is a symbol that instructs 'C' compiler to perform some operation, or action, on one or more operands. Operators are used in programs to manipulate data and variables. The operations generally involve arithmetic or logical manipulations.

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

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Unary Operators
6. The Conditional Operator
7. The Comma Operator
8. The Bitwise operators

1. Arithmetic Operators

The Arithmetic Operators allows in 'C' are +, -, *, / and the modulus operator %. These operators can be applied to any data type available in 'C'.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division

The '%' operator is referred to as the modulus operator. It returns the remainder, when a number is divided with another number. Let a and b are two variables holding integers values 10, 3 respectively. The above operators work as follows.

Expression	Int Value
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

/* Program to explain Arithmetic operators */

```
# include <stdio.h>
main( )
{
    int a = 10, b = 30;
    printf("Addition = %d\n", b+a);
    /* Performing addition */
    printf("Subtraction = %d\n", b-a);
    /* Performing Subtraction */
    printf("Division = %d\n", b/a);
    /* Performing Division */
    printf("Multiplication = %d\n", b*a);
    /* Multiplication */
    printf("Remainder = %d\n", b%a);
    /* Gives the remainder */
}
```

Output :

```
Addition    = 40
Subtraction  = 20
Division     = 3
Multiplication = 300
Remainder    = 0
```

2. Relational Operators

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<', meaning 'less than'. An expression such as

a < b or 1 < 20

containing a relational operator is termed as a *relational expression*. The value of a relational 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

C supports six relational operators in all.

These operators and their meanings are shown in table below.

Relational Operators	
Operator	Meaning
<	is less than
< =	is less than or equal to
>	is greater than
> =	is greater than or equal to
= =	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

ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values.

4.5 < = 10 TRUE

4.5 < - 10 FALSE

-3.5 > = 0 FALSE

10 < 7 + 5 TRUE

a + b = c + d TRUE only if the sum of values of a and b is equal to the sum of value of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators.

Relational expressions are used in *decision statements* such as if and while to decide the course of action of a running program.

3. Logical Operators

In addition to the relational 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*. Like the simple relational expressions, a logical expression also yields a value of *one* or *zero*, according to the truth table shown in Table below. The logical expression given above is true only if *a > b* is *true* and *x == 10* is *true*. If either of them are false, the expression is *false*.

op-1	op-2	Value of the expression	
		op-1 && op-2	op-1 op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Some examples of the usage of logical expressions are :

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

Example :

```
#include<stdio.h>
main( )
{
    int a=10, b=20;
    printf("%d/n", a<b);    /* if the condition is true return 1 otherwise return 0 */
    printf("%d/n", a>b);
    printf("%d/n", a== b);
    printf("%d/n", a>=b);
    printf("%d/n", a<= b && b>a); /* if both conditions are true return 1 otherwise 0*/
}
```

Output :

```
1
0
0
0
1
```

4. Assignment Operator

The assignment operator is the equal sign (=). The assignment expressions that make use of this operator are written in the form.

Syntax :

Identifier = Expression

where identifier generally represents a variable and expression represents a constant, a variable, or a more complex expression.

$x = y$ → 'assign the value of y to x'
 $a = 3$ → a integer vale 3 is assigned to the variable a
 $sum = a + b$ → value of a + b is assigned to sum
 $delta = 0.001$ → floating point value is assigned to delta

Example :

```

/* Program to explain assignment operator */
#include<stdio.h>
main( )
{
    int a = 10;    /* '=' is assignment operator */
    int b=c=a,d;   /* a is assigned to c is assigned to a */
    d=(a+b+c)/3;  /* Adding three values, dividing by 3, result is assigned to d */
    printf("The values are \na\%d/nb=%d/nc=%d/n", a, b, c);
    print("The resultant value = %d/n", d);
}

```

Output :

The values are
 a = 10
 b = 10
 c = 10
 The resultant value = 10

5. Unary Operators

The class of operators in 'C' which act upon a single operand to produce a new value are called Unary Operators.

- The most commonly used unary operator is unary minus, where a numeric constant, variable or expression is preceded by a minus sign.
- The unary operator predefined to an integer constant tells the compiler to reverse the sign by subtracting the value from zero.

Example :

-339 $-8E-8$ $-(x + y)$

The other commonly used unary operators are

- 1) Increment operator
- 2) Decrement operator
- 3) Size of operator

(i) The Increment Operator (+ +)

This operator increments the operand by one.

$++X$ is equivalent to $X = X + 1$

$X++$ is equivalent to $X = X + 1$

Example :

```
/* Program to explain increment operator */
#include<stdio.h>
main( )
{
    int i = 10;
    printf("i value is %d\n", i);
    /* incrementing i value (post increment)*/
    i++;
    printf("After post increment i values is = %d\n", i);
    /* incrementing i value (pre increment) */
    ++i;
    printf("After pre increment i value is = %d\n", i);
}
```

Output :

```
i value is 10
After post increment i value is = 11
After pre increment i value is = 12
```

(ii) Decrement operator (--)

This operator decrements the operand by one.

-- X is equivalent to X = X - 1

X -- is equivalent to X = X - 1

Example :

```
/* Program to explain decrement operator */
#include<stdio.h>
main( )
{
    int j = 10;
    printf("j value is = %d\n", j);
    /* Decrementing j value (post decrement) */
    j--;
    printf("After post decrement j value is = %d\n", j);
    /* Decrementing j value (pre decrement) */
    --j;
    printf("After pre decrement j value is = %d\n", j);
}
```

Output :

```
j value is 10
After post decrement j value is = 9
After pre decrement j value is = 8
```

- The Increment and Decrement Operators can be used only with variables, not with constants. The operation performed is to add one to, or subtract one from the operand. The above two operations can each be utilized in two different ways depending on whether the operator is written before or after the operand.
- If the operator precedes the operand (e.g. ++X) then the operand will be altered before it is utilized within the program.
- If the operator follows the operand (e.g. X++) then the value of the operand will be altered after it is utilized.

```
x = 10;  
y = x++;
```

After these statements are executed, x has value 11, and y has value 10. The value of x was assigned to y and then x was incremented.

```
x = 10;  
y = ++x
```

After these statements are executed, both x, y have the value 11. Since first x is incremented and then it is assigned to y.

- Thus in above two statements, the ++ in x++ is **post increment** operator, While that in ++x is called the **pre increment** operator.
- Thus depending upon the condition and statements required, the unary operator is placed either before its operand (prefix mode) or after its operand (post fix mode).

Example :

Program : Illustration increment & decrement operators

```
#include<stdio.h>  
main( )  
{  
    int i = 10, j = 10;  
    printf("i value is = %d\n", i);  
    printf("j value is = %d\n", j);  
  
    /* i++ means post increment */  
    printf("\n incrementing \n\n");  
    printf("\n post increment of i= %d", i++);  
    printf("\n now i value    = %d", i);  
  
    /* ++j means pre increment */  
    printf("\n pre increment of j = %d", ++j);  
    printf("\n now j value    = %d", j);  
    printf("\n\n i value is    = %d\n", i);  
    printf("j value is = %d\n", j);  
  
    /* i-- means post decrement */  
    printf("\n decrementing \n\n");  
    printf("\n post decrement of i = %d", i--);
```

```

printf("\n now i value      = %d", i);
printf("\n pre decrement of j = %d", --j);
    // --j means pre decrement
printf("\n Now j value = %d", j);
}

```

Output :

```

i value is    = 10
j value is    = 10
incrementing
post increment of i =10
now i value    = 11
pre increment of j = 11
now j value    = 11
i value is     = 11
j value is     = 11
decrementing
post decrement of i = 11
now i value     = 10
pre decrement of j = 10
Now j value = 10

```

(iii) size of operator

sizeof operator is used to return the sizeof operand, in bytes. This operator always precedes its operand, which is an expression. sizeof operator helps to know the number of bytes allocated in the memory to various types of data items.

Example :

```

/* Program to explain sizeof operator */
#include<stdio.h>
main( )
{
    int i;
    float f;
    char c;
    double d;
    long double ld;
    printf("\sizeof integer = %d\n", sizeof(i));
    printf("\sizeof float   =%d\n",sizeof(f));
    printf("\sizeof character = %d\n", sizeof(c));
    printf("\sizeof double   = %d\n", sizeof(d));
    printf("\sizeof long double = %d\n", sizeof(ld));
}

```

Output :

```

sizeof integers    = 2
sizeof float       = 4
sizeof character    = 1
sizeof double      = 8
sizeof long double = 10

```


6. The Conditional Operator

The Conditional Operator `?` and `:` are sometimes called **Ternary Operator** since they take three arguments at a time. An expression that makes use of the conditional operator is called conditional expression.

Syntax : Expression 1 ? Expression 2 : Expression 3

When evaluating, exp 1 is evaluated first, if exp 1 is true, then exp 2 is evaluated and this becomes the value of the conditional expression. However, if expression 1 is false, then exp 3 is evaluated and it becomes the value of the conditional expression.

`x = 16` are int variables

`y = (x > 5 ? 3:4)`

This statement will store 3 in y, if x is greater than 5, otherwise it will store 4 in y.

Example :

```
/* Program using Relational, Logical, conditional (?:) operators to find biggest of
two numbers */
#include<stdio.h>
main( )
{
    int a = 10, b = 20, big;
    big=(a>b)? a:b;    /* ?: is the conditional operator.
                       > is relational operator */
    printf("Biggest=%d", big);
}
```

Output :

Biggest = 20

7. Comma Operator

The comma operator `( , )` allows two or more different expressions to appear when only one expression is expected. The expressions are separated by the comma operator. The comma operator takes its arguments and evaluates them from left to right and returns the value of the rightmost expression.

Consider the following statement :

`i = (j = 10, k = 5, j + k);`

Here, the operator first evaluates '`j = 10`' which assigns 10 to the variable j. Then the next expression '`k = 5`' is evaluated which assigns 5 to k. The third expression is evaluated and its value returned as the operator's result. Thus, i is assigned 15.

8. Bitwise Operator

C supports a set of bitwise operations. The lowest logical element in the memory is **bit**. Bit manipulation is a method of working at the bit level with the computer system. **C** allows the programmer to interact directly with the hardware of a particular system through bitwise operators and expression. These operators work only with **int** and **char** datatypes and cannot be used with **float** and **double** type.

The table below shows the bitwise operators that are available in C.

Operator	Meaning
~	One's complement
	Bitwise OR
&	Bitwise AND
^	Bitwise Exclusive OR
>>	Right shift
<<	Left shift

Examples :

```

z = x & y;    /* x AND y */
z = x | y;    /* x OR y */
z = ~z       /* Bit complement */
z = x ^ y;    /* x XOR y */
z = x << y;   /* Left shift : shift x left by number of bits stored in y */
z = x >> y;   /* Right shift : shift x right by number of bits stored in y */

```

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

```
int main()
```

```
{
```

```
    unsigned char a = 5, b = 9; // a = 5(00000101), b = 9(00001001)
```

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

```
    printf("a&b = %d\n", a&b); // The result is 00000001
```

```
    printf("a|b = %d\n", a|b); // The result is 00001101
```

```
    printf("a ^ b = %d\n", a ^ b); // The result is 00001100
```

```
    printf("~a = %d\n", a = ~a); // The result is 11111010
```

```
    printf("b<<1 = %d\n", b<<1); // The result is 00010010
```

```
    printf("b>>1 = %d\n", b>>1); // The result is 00000100
```

```
    return 0;
```

```
}
```

Output:

```
a = 5, b = 9
```

```
a&b = 1
```

```
a|b = 13
```

```
a ^ b = 12
```

```
~a = 250
```

```
b<<1 = 18
```

```
b>>1 = 4
```

1.27 EXPRESSION EVALUATION

Q51. What are expression evaluation?

Ans : (June-19, June-18)

It is important to note that the above section on operator precedence and associativity does not define the order that the operands are evaluated; it only defines how an expression is interpreted. The operands to most operators may be evaluated in any order.

For example the simple expression $a * b + c$ is interpreted as $(a * b) + c$ according to the precedence rules. But that does not mean $a * b$ is evaluated first. The c variable could be evaluated first, then $a * b$, which in turn could evaluate either a or b .

When dealing with simple variables this is not a problem. However consider what happens if the operands to an operator have side effects:

```

1.  #include <stdio.h>
2.  int a = 0;
3.  int func1()
4.  {
5.      a *= 3;
6.      return a;
7.  }
8.  int func2()
9.  {
10.     a += 3;
11.     return a;
12. }
13. void main()
14. {
15.     int b = func1() + func2();
16.     printf("a = %d, b = %dn", a, b );
17. }
```

In this example both functions **func1()** and **func2()** modify the variable **a** and return its value. The output of the program depends on which order these functions are called on line 15. If **func1()** is called first it will assign 0 to **a** and return 0, then when **func2()** is called it will assign 3 to **a** and return 3. This leaves **a** and **b** both set to 3.

If **func2()** is called first, it will set **a** to 3 and return 3, then when **func1()** is called it will set **a** to 9 and return 9. This leaves **a** set to 9 and **b** set to 12.

Both outputs would be valid since C does not specify which operand should be evaluated first. C also does not specify in which order arguments to functions are evaluated. If in the previous example we had a function like this :

```

int func3( int arg1, int arg2 )
{
    return arg1 + arg2;
}
```

and we changed line 15 to:

```
int b = func3( func1(), func2() );
```

We still have the same problem. Either **func1()** or **func2()** could be called first which will change the arguments passed to **func3()**.

These examples have emphasized the fact that the order of evaluation of operands may not be what you expect. You must be especially careful with expressions that cause side effects. They could cause situations where the program works in one environment but fails in another.

1.28 PRECEDENCE AND ASSOCIATIVITY

Q52. Explain the precedence and Associativity of operators?

Ans : (June-19, June-18, Imp.)

Hierarchy (Precedence) of Operators

The priority or precedence in which the operation of an arithmetic statement are performed is called the **hierarchy of operations**.

Each operator has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated.

There are distinct levels of precedence and an operator may belong to one of the levels. The operator at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the **associativity property** of an operator. The precedence of commonly used operators are shown in the table under the heading precedence of operator.

Consider the following expression.

$$a = 3 * 4/5 + 10/5 + 8 - 1 + 7/8$$

Step evaluation of above expression is shown below.

	Operation
$a = 12/5 + 10/5 + 8 - 1 + 7/8$	*
$= 2 + 10/5 + 8 - 1 + 7/8$	/
$= 2 + 2 + 8 - 1 + 7/8$	/
$= 2 + 2 + 8 - 1 + 0$	/
$= 4 + 8 - 1 + 0$	+
$= 12 - 1 + 0$	+
$= 11 + 0$	-
$= 11$	+

Note that 12/5, 10/5 and 7/8 are all integer divisions, hence 7/8 evaluates to zero.

Operator	Description	Associativity	RANK
()	function call	Left to right	1
[]	Subscript (Array element reference)	"	1
->	Element selector	"	1
!	Boolean NOT (Logical Negation)	Right to left	2
~	Binary NOT	"	2
+	Increment	"	2
--	Decrement	"	2
-	Unary Minus	"	2
+	Unary Plus	"	2
*	Indirection (Dereference)	"	2
&	Address	"	2
Size of	Size of an object	"	2
(type)	Type cast (Conversion)	"	2
*	Multiplication	Left to right	3
/	Division	"	3
%	Modulo	"	3
+	Addition	"	4
-	Subtraction	"	4
<<	Left Shift	"	5
>>	Right Shift	"	5
<	Less than	"	6
<=	Less than or equal to	"	6
>	Greater than	"	6
>=	Greater than or equal to	"	6
==	Equality	"	7
!=	Not Equal to	"	7
&	Binary AND	"	8
^	Binary XOR	"	9
	Binary OR	"	10
&&	Boolean AND	"	11
	Boolean OR	"	12
?:	Conditional	Right to Left	13
=	Assignment	"	14
op=	Operator assignment(+=, -=, *=, /=, %=)	"	14
,	Comma	Left to Right	15

1.29 TYPE CONVERSIONS

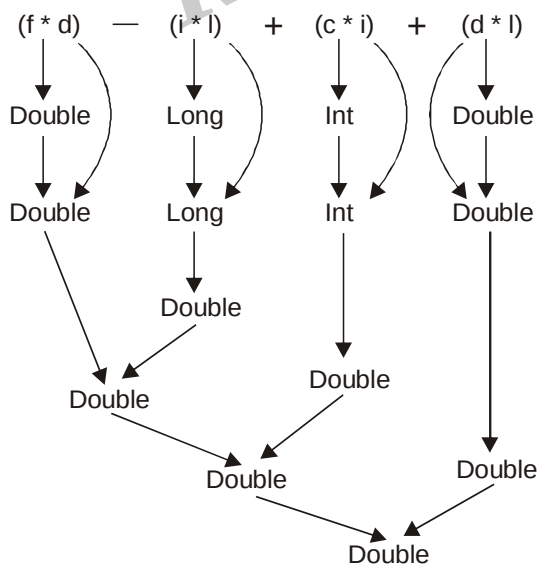
Q53. What is conversion? Explain different types with examples?

Ans : (Nov.-20)

When constants and variables of different types are mixed in an expression they are converted to the same type. This conversion is done implicitly by C compiler. The C compiler will convert all operands to the type of the largest operand. For example, if an expression contains an operation between an int and a float, the int would be automatically promoted to a float before carrying out the operation. The conversion rules are :

1. All the chars and shortints are converted to ints. All floats are converted to double.
2. For all operand pairs : If one of the operands is double the other operand is converted to double. If one of the operands is long, the other operand is converted to long.

If one or the operand is unsigned the other is converted to unsigned. In an expression, all the types mixed, the rules must be applied in sequence : unsigned, long, double.



Example :

```

char c ;
int i;
double d;
float f;
long int l ;
  
```

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper types so that the expression can be evaluated without losing any significance. This automatic conversion is known as *implicit type conversion*.

Below is the sequence of rules that are applied while evaluating expression. All **short** and **char** are automatically converted to **int**; then

1. If one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double**;
2. else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
3. else, if one of the operands, is **float**, the other will be converted to **float** and the result will be **float**;
4. else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;
5. else, if one of the operands is **long int** and the other is **unsigned int**, then
 - a) if **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted a such and the result will be **long int**;
 - b) else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int** ;

6. else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int**;
7. else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int**.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. The following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

Explicit Conversion

There are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

```
ratio = female_number/male_number
```

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

```
ratio = (float) female_number/male_number
```

The operator **(float)** converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator **(float)** affect the value of the variable **female number** - and the type of **female number** remains as **int** in the other parts of the program.

The process of such a local conversion is known as *explicit conversion* or *casting a value*. The general form of a cast is :

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

where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression.

Program below illustrates the use of cast to evaluate the equation SUM =

```
main( )
{
    float sum ;
    int  n ;
    sum = 0 ;
    for(n = 1; n<=0; ++n)
    {
        sum = sum + 1/(float)n ;
        printf("%2d %6.4f\n", n, sum) ;
    }
}
```

Type Casting

A type cast provides a method for converting a variable to a particular type. Type casting force an expression to be of a specific data type by using a construct called a cast.

The general form of a cast is

(type) expression

That is the data type to which the value of original expression is converted and is written in parenthesis to the left of the expression. For example, if x is originally declared as a type int, then to convert it to a type float.

y = (float) x

The variable y will now contain the value of x converted to float.

Here the type and value x are unchanged. Generally, casts are often considered as operators. Cast operator is unary operators. Casting is very useful when to display a real value after integer division.

i.e., (float)x/w

Here, cast will not change the value of x, but it produces a real result. The cast operator cannot be used on a type void. C allows to cast any other type.

Rahul Publications

Short Question & Answers

1. Define Computer.

Ans :

Computers have touched every part of our lives: the way we work, the way we learn, the way we live, even the way we play. It almost is impossible to go through a single day without encountering a computer, a device dependent on a computer, information produced by a computer, or a word that was introduced or whose meaning has changed with the advent of computers. Because of the significance of computers in today's world, it is important to be computer literate. Being computer literate means you have knowledge and understanding of computers and their uses.

Computer is a Powerful Tool

A computer is a powerful tool because it is able to perform the information processing cycle operations (input, process, output, and storage) with amazing speed, reliability, and accuracy; store huge amounts of data and information; and communicate with other computers. Computers allow users to generate correct information quickly, hold the information so it is available at any time, and share the information with other computer users.

Definition

A computer is a machine that manipulates data according to a list of instructions.

Computer is a modern electronic tool that accepts input data, do processing according to a set of instructions, called program and produce/store the output information. The term computer has been derived from the word 'COMPUTE'. This computer generally gives impression that it is a calculating machine but really computer can do several jobs other than computation.

2. List out the capability of computer.

Ans :

Capabilities of a Computer

Computers are versatile tools with seemingly limitless capabilities. These capabilities are categorized depending on their type and application. Some of the common capabilities of computers are given below :

Computers are capable of :

1. Performing tasks (or computations) at very high speed.
2. Performing computations with desired accuracy.
3. Receiving the data and producing useful information.
4. Storing instructions and data for future processing or use.
5. Executing a set of instructions (called program) with or without manual intervention in a well-defined manner.
6. Supporting decision making by using logical rules.

7. Communicating with other systems, even those at remote places.
 8. Exploiting a complex internal structure in a variety of ways.
 9. Performing the tasks (or subtasks) repetitively.
-

3. Explain different components of computers.

Ans :

A computer can accept input, process or store data, and produce output according to a set of instructions which are fed into it. The two main characteristics of a computer are: (i) it responds to a specific set of instructions in a well-defined manner. (ii) It can execute a pre-recorded list of instructions (a program). Modern computers are electronic and digital. The actual machinery - wires, transistors and circuits is called hardware; the instructions and data are called software.

All general purpose computers require the following hardware components :

- i) **Central Processing Unit (CPU):** The 'brain' of the computer, the component that actually executes instructions.
 - ii) **Memory :** It enables a computer to store, at least temporarily, data and programs.
 - iii) **Input device :** Usually a keyboard or mouse is used to read data and programs into the computer.
 - iv) **Output device :** A display screen, printer, etc. that lets you see what the computer has accomplished.
 - v) **Mass storage device :** It allows a computer to permanently store large amounts of data. Common mass storage devices include disk drive and tape drive.
-

4. What is cache memory? Explain its advantages & disadvantages.

Ans :

Cache memory is a very high speed semiconductor memory which can speed up CPU. It acts as a buffer between the CPU and main memory. It is used to hold those parts of data and program which are most frequently used by CPU. The parts of data and programs are transferred from disk to cache memory by operating system, from where CPU can access them.

Advantages

- The advantages of cache memory are as follows:
- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

Disadvantages

- The disadvantages of cache memory are as follows:
- Cache memory has limited capacity.
- It is very expensive.

5. Explain about RAM.

Ans :

Random Access Memory

Random-access memory (usually known by its acronym, **RAM**) is a computer data storage. Today it takes the form of integrated circuits that allow the stored data to be accessed in any order, i.e. at random. The word random thus refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether or not it is related to the previous piece of data

RAM is temporary and volatile; that is RAM chips lose their contents if the current is lost or turned off. RAM chips are located directly on the computer's main circuit board or in other chips located on peripheral cards that plug into the main circuit board.

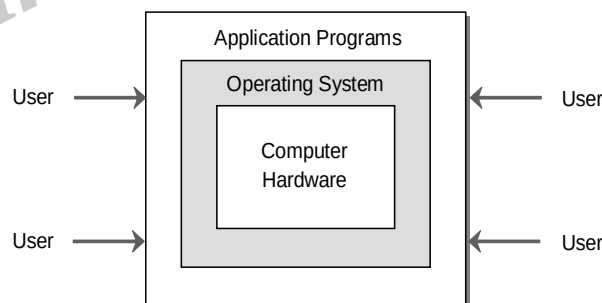
The two main types of RAM are dynamic RAM (DRAM) and static RAM (SRAM). DRAM memory chips offer the greatest capacities and the lowest cost per bit, but are relatively slow. SRAM costs more than DRAM but has a higher level of performance, making SRAM the preferred choice for performance-sensitive applications, including the external L2 and L3 caches that speed up microprocessor performance.

6. Define Operating System and it's types.

Ans :

An **operating system** is an interface between hardware and user. An OS is responsible for the management and coordination of activities and the sharing of the resources of the computer. The operating system acts as a host for computing applications run on the machine.

An operating system is a set of programs which act as an interface between the user of computer and the computer hardware, users or applications programs use the hardware indirectly through operating system as shown in figure.



The operating system controls and coordinates the use of hardware among the application programs. The primary aim of an operating system is to provide convenience to the user in using the system, and the secondary aim is to use the system hardware in an efficient way. The OS is used to manage the various resources and operations of the computer system.

Operating systems can be classified as follows :

1. **Multi-user** : Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
2. **Multiprocessing** : Supports running a program on more than one CPU.

3. **Multitasking** : Allows more than one program to run concurrently.
 4. **Multithreading** : Allows different parts of a single program to run concurrently.
 5. **Real time: Responds to input instantly.** General-purpose operating systems, such as DOS and UNIX, are not real-time.
-
7. **Define ALU and CU.**

Ans :

1. **Arithmetic and Logic Unit (ALU)**

The ALU performs arithmetic and logical operations on data such as addition, subtraction, multiplication, division, comparison, AND, OR, NOT etc. ALU receives data as bits (a smallest unit of data) and executes the processing operations under the direction of program instructions. Most of the execution part within a computer is carried out by ALU.

2. **Control Unit**

The control unit manages and coordinates the entire computer system. It generates control and timing signals necessary for the execution of instructions. This unit directs the specific instruction and data that are to be processed and determines the correct order and sequence. The control unit obtains instructions from the program stored in the main memory, interprets the instruction and issues signals that cause other units of the system to execute them.

8. **List the Benefits of HLL.**

Ans :

Advantages of High-level Languages

1. The high-level languages are easier to use and understand than the machine language and the assembly language.
2. A high-level language is more expressive than the low-level language and hence the programs written in high-level language are much more compact and self explanatory than their low-level counterparts.
3. High-level languages provide a better or superior documentation than the low-level languages.
4. They are totally independent of the machine on which they are being used. This makes it possible for the users to use the same programs written on one machine, on different machine making the programs portable.
5. Correcting and modifying programs is much easier than low-level languages.

9. **What is the use of Compiler?**

Ans :

Compiler

A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program.

Compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. More generally, compilers are a specific type of translator.

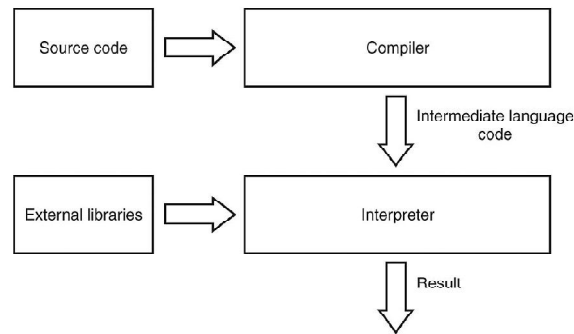


Fig.: Compiler action

During the process of translation, the compiler reads the source program statement-wise and checks for syntax errors. In case of any error, the computer generates a printout of the same. This action is known as diagnostics.

10. What are the differences between Compiler & Interpreter

Ans :

Compiler	Interpreter
1. Compiler Takes Entire program as input.	1. Interpreter Takes Single instruction as input.
2. Intermediate Object Code is Generated	2. No Intermediate Object Code is Generated.
3. Conditional Control Statements are Executes faster.	3. Conditional Control Statements are Executes slower.
4. Memory requirement is more. (Since Object Code is Generated)	4. Memory requirement is less.
5. Program need not be compiled every time.	5. Every time higher level program is converted into lower level program.
6. Errors are displayed after entire program is checked.	6. Errors are displayed for every instruction interpreted (if any).
7. Example : C & C++	7. Example : BASIC, Python, Ruby

11. Define Algorithm and list out it's characteristics.

Ans :

Algorithm is a finite sequence of explicit and unambiguous steps required to solve the given problem. When the required input values are provided, it produces an output and then terminates. The steps may repeat or require decisions as per the requirement.

The same problem can be solved with different methods. So, for solving the same problem, different algorithms may be accomplished. In these algorithms, the steps, time, space and efforts may vary more or less. User writes algorithm in his/her own language. So it cannot be executed on computer.

Characteristics of Algorithms

Algorithms are not computer programs, as they cannot be executed by a computer. The characteristics of a good algorithm are :

- **Precision** – the steps are precisely stated (defined).
- **Uniqueness** – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- **Finiteness** – the algorithm stops after a finite number of instructions are executed.
- **Input** – An algorithm has zero or more inputs, taken from a specified set of objects.
- **Output** – An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Generality** – the algorithm applies to a set of inputs.
- **Effectiveness** – All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

12. Define flowcharts.

Ans :

A flowchart is a visual representation of the sequence of steps and decisions needed to perform a process. Each step in the sequence is noted within a diagram shape. Steps are linked by connecting lines and directional arrows. This allows anyone to view the flowchart and logically follow the process from beginning to end.

The flowchart has different shapes, there are two shapes: those with rounded ends represent the start and end points of the process and rectangles are used to show the interim steps. These shapes are known as flowchart symbols. Table below shows some standard symbols which are frequently required in the flowcharts.

13. Define pseudocode.

Ans :

Pseudocode is made up of two words : pseudo and code. Pseudo means imitation and code refers to instructions, written in a programming language. As the name suggests, pseudocode is not a real programming code, but it models and may even look like programming code. Pseudocode is an outline of a program, written in a form that can be easily converted into real programming statements. Pseudocode uses plain English statements rather than symbols, to represent the processes of a computer program.

Pseudocode strikes a fine balance between the understandability and informality of a natural language like English, and the precision of a computer program code. It is somewhat halfway between English and a programming language. If an algorithm is written in English, the description may be at such a high level that it may prove difficult to analyze the algorithm and then to transform it into actual code. The goal of writing pseudocode is to provide a high-level description of an algorithm, which facilitates analysis and eventual coding, but at the same time suppresses many of the details that are insignificant.

14. What is the use of C-preprocessor?

Ans :

The **C-preprocessor (cpp)** is the preprocessor for the C programming language. In many C implementations, it is a separate program invoked by the compiler as the first part of translation. The preprocessor handles directives for source file inclusion (`#include`), macro definitions (`#define`), and conditional inclusion (`#if`). The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other types of files.

Most C preprocessor features are inactive unless you give specific commands to request their use. But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands.

- All C comments are replaced with single spaces.
- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions.

15. Define C-Token.

Ans :

In a C program the smallest individual units are known as C tokens. A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. Tokens are basic building blocks of C Programming.

C has six types of tokens as shown in fig.below. C programs are written using these tokens are the syntax of the language.

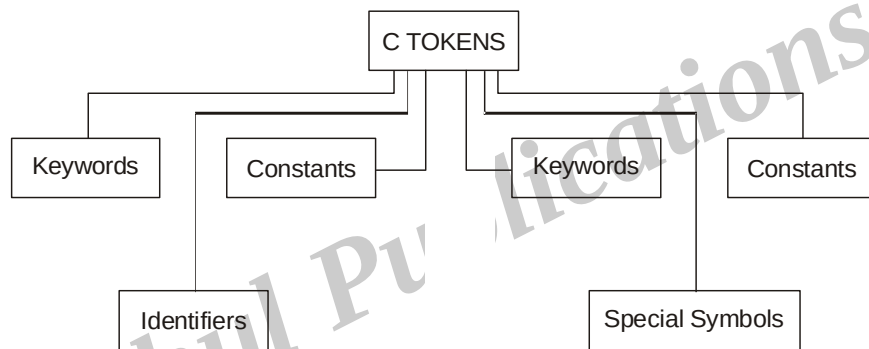


Fig.: C tokens and examples

16. Write the differences between Identifier and Keyword.

Ans :

Identifier vs Keyword

Identifier	Keyword
Identifiers are names that are given to various program elements, such as variable, function and arrays.	C take some reserved word called keyword, they have predefine meaning in C.
Identifiers consist of letters and digits.	Keyword consist only letter.
Identifier's first character must be a letter.	Keyword's all character is letter.
Identifiers Upper & lowercase letter is use.	Keywords are all lowercase.
Upper and lowercase are not equivalent.	Upper and lowercase are also not equivalent.
Like: X, sum_5, _weather etc. But 4th is not identifier cause identifier first character must a letter.	Like: auto, short, long etc.

17. Define variable.

Ans :

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.

18. Explain size of operator.

Ans :

Size of Operator

sizeof operator is used to return the sizeof operand, in bytes. This operator always precedes its operand, which is an expression. sizeof operator helps to know the number of bytes allocated in the memory to various types of data items.

Example :

```
/* Program to explain sizeof operator */
#include <stdio.h>
main( )
{
    int i;
    float f;
    char c;
    double d;
    long double ld;
    printf("sizeof integer = %d\n", sizeof(i));
    printf("sizeof float = %d\n", sizeof(f));
    printf("sizeof character = %d\n", sizeof(c));
    printf("sizeof double = %d\n", sizeof(d));
    printf("sizeof long double = %d\n", sizeof(ld));
}
```

Output :

```
sizeof integers      = 2
sizeof float         = 4
sizeof character     = 1
sizeof double        = 8
sizeof long double   = 10
```

19. Explain conditional operation.

Ans :

The Conditional Operator

The Conditional Operator ? and : are sometimes called **Ternary Operator** since they take three arguments at a time. An expression that makes use of the conditional operator is called conditional expression.

Syntax : Expression 1 ? Expression 2 : Expression 3

When evaluating, exp 1 is evaluated first, if exp 1 is true, then exp 2 is evaluated and this becomes the value of the conditional expression. However, if expression 1 is false, then exp 3 is evaluated and it becomes the value of the conditional expression.

x = 16 are int variables

y = (x > 5 ? 3:4)

This statement will store 3 in y, if x is greater than 5, otherwise it will store 4 in y.

Example :

```
/* Program using Relational, Logical, conditional (?) operators to find biggest of
two numbers */
#include<stdio.h>
main( )
{
    int a = 10, b = 20, big;
    big=(a>b)? a:b;    /* ?: is the conditional operator.
                       > is relational operator */
    printf("Biggest=%d", big);
}
```

Output :

Biggest = 20

20. What is type conversion?

Ans :

When constants and variables of different types are mixed in an expression they are converted to the same type. This conversion is done implicitly by C compiler. The C compiler will convert all operands to the type of the largest operand. For example, if an expression contains an operation between an int and a float, the int would be automatically promoted to a float before carrying out the operation. The conversion rules are :

1. All the chars and shortints are converted to ints. All floats are converted to double.
2. For all operand pairs : If one of the operands is double the other operand is converted to double. If one of the operands is long, the other operand is converted to long.

If one or the operand is unsigned the other is converted to unsigned. In an expression, all the types mixed, the rules must be applied in sequence : unsigned, long, double.

Choose the Correct Answers

1. Which of the following converts a C program's source code into machine language: [a]
(a) compiler (b) Editor
(c) operating system (d) none of the above
2. The operating system manages [d]
(a) Memory (b) Processor
(c) Disk and I/O devices (d) All of the above
3. Which of the following is not a valid variable name declaration? [d]
(a) int __a3; (b) int __3a;
(c) int __A3; (d) None of the mentioned
4. Which of the following cannot be a variable name in C? [a]
(a) volatile (b) true
(c) friend (d) export
5. The correct order of evaluation for the expression "z = x + y * z / 4 % 2 - 1" [d]
(a) $- * / \% = + -$ (b) $- / * \% - + =$
(c) $- + = * \% /$ (d) $- * / \% + - =$
6. What is the size of an int data type? [d]
(a) 4 Bytes (b) 8 Bytes
(c) Depends on the system/compiler (d) Cannot be determined
7. Which of the datatypes have size that is variable? [b]
(a) int (b) struct
(c) float (d) double
8. Which of the following declaration is not supported by C? [a]
(a) String str; (b) char *str;
(c) float str = 3e2; (d) Both (a) and (c)
9. What is the value of x in this C code? [c]

```
#include <stdio.h>
void main( )
{
int x = 5*9/3+9
}
```


(a) 3.75 (b) Depends on compiler
(c) 24 (d) 3
10. Relational operators cannot be used on: [a]
(a) structure (b) long
(c) strings (d) float

11. Result of a logical or relational expression in C is [b]
(a) True or False
(b) 0 or 1
(c) 0 if expression is false and any positive number if expression is true
(d) None of the mentioned
12. What is the type of the below assignment expression if x is of type float, y is of type int? [a]
`y = x + y;`
(a) int
(b) float
(c) There is no type for an assignment expression
(d) double
13. Operation "`a = a * b + a`" can also be written as: [d]
(a) `a *= b + 1;`
(b) `(c = a * b) != (a = c + a);`
(c) `a = (b + 1) * a;`
(d) All of the mentioned
13. In expression `i = g() + f()`, first function called depends on [a]
(a) Compiler
(b) Associativity of `()` operator
(c) Precedence of `()` and `+` operator
(d) Left to right of the expression
14. What is the output of this C code ? [c]
`#include <stdio.h>`
`void main()`
`{`
`double b = 5 % 3 & 4 + 5 * 6;`
`printf("%lf", b);`
`}`
(a) 2 (b) 30
(c) 2.000000 (d) Run time error
15. Which of the following are unary operators? [d]
(a) `sizeof` (b) `-`
(c) `++` (d) All of the mentioned
16. Where in C the order of precedence of operators do not exist? [d]
(a) Within conditional statements, if, else (b) Within while, do-while
(c) Within macro definition (d) None of the mentioned

17. Associativity of an operator are: [d]
(a) Right to Left (b) Left to Right
(c) Random fashion (d) Both (a) and (b)
18. Which of the following method are accepted for assignment? [b]
(a) $5 = a = b = c = d$; (b) $a = b = c = d = 5$;
(c) $a = b = 5 = c = d$; (d) None of the mentioned
19. Which of the following is NOT possible with any 2 operators in C? [-]
(a) Different precedence, same associativity
(b) Different precedence, different associativity
(c) Same precedence, different associativity.
(d) All of the mentioned
20. Which of the following is possible with any 2 operators in C? [d]
(a) Same associativity, different precedence
(b) Same associativity, same precedence
(c) Different associativity, different precedence
(d) All of the mentioned

Rahul Publications

Fill in the Blanks

1. A c program is basically a collection of _____
2. C language is well suited for _____ programming.
3. A _____ character instructs the computer to move the control to the next line.
4. C program execution begins from _____
5. Local variable which exists and retains its value even after the control is transferred to the calling function is, _____ storage class.
6. The operator “+ +” is know as _____ operator.
7. The _____ operator can be used to determine the length of array and structures.
8. The standard mathematical functions are included in the _____ header file.
9. _____ function can be used to read a single character.
10. Logical and is performed with _____ operator.

ANSWERS

1. functions
2. Structure
3. newline
4. main ()
5. static
6. Increment
7. size of
8. math.h
9. getch()
10. &&

UNIT II

Input-Output: Non-formatted and Formatted Input and Output Functions, Escape Sequences, Control Statements: Selection Statements - if, if-else, nested if, nested if-else, comma operator, conditional operator, switch; Iterative Statements - while, for, do-while; Special Control Statement - goto, break, continue, return, exit. Arrays and Strings: One-dimensional Arrays, Character arrays, Functions from ctype.h, string.h, Multidimensional Arrays.

2.1 INPUT AND OUTPUT OPERATIONS

Q1. What are I/O operators & their functions ?

Ans :

C provides several functions that give different levels of input and output capability. These functions are, in most cases, implemented as routines that call lower-level input/output functions.

Though, C has no built in I/O statements for receiving data from input devices and sending data to the output devices, C language offers facility for inputting and outputting data from and to the files and devices. This is done by writing a few small programs that would link the C compiler for particular operating system's I/O facilities. Thus, the C language manage its inputting and outputting functions by developing several standard I/O functions and compiling them into the C library functions. The C language developer has standardized header files, which contain these several functions, developed to manage the I/O functions.

Few commonly used header files in C language are given below :

<stdio.h> Standard I/O library functions

<math.h> Mathematical functions

<ctype.h> Character testing and conversion functions

<string.h> String manipulation functions

<time.h> Time manipulation functions

A number of functions and macros exist to provide support for streams of various kinds. The <stdio.h> header file contains the various declarations necessary for the functions, together with the macros and type declarations needed for the input and output functions. The input/output functions fall into two categories: non-formatted read (input) and display (output) functions and formatted read (input) and display (output) functions.

C does not have any built-in input/output statements as part of its syntax. All the standard input/output operation is carried out through function calls such as printf and scanf. There exist several functions

that have more or less become standard for input and output operations in C. These functions are known as standard input/output library. Few types of input/output functions are as follows :

Input Functions	Output Functions
scanf()	printf()
getchar()	putchar()
gets()	puts()
getche()	putch()
fgets()	

Here, the gets() and puts() functions are used to get long input and long output strings. In the same way, the getchar() and putchar() functions are used to get a single character output.

2.1.1 Non-formatted / formatted Input/ Output Operations

Q2. Explain non-formatted & formatted I/O operations ?

Ans : (June-19, Dec.-18, June-18, Imp.)

1. Non-formatted I/O operations

Non-formatted input/output is the simplest and most efficient form of input/output. It is usually the most compact way to store data. Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation.

Non-formatted input and output can be carried out by standard input-output library functions in C. These can handle one character at a time. For the input functions it does not require <Enter> to be pressed after the entry of the character. For output functions, it prints a single character on the console.

To use input and output functionality in your program, you need to include stdio header :

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

Character I/O : getchar and putchar

getchar reads a character from the standard input device, while **putchar** writes a character to the standard output device.

The getchar takes the following form :

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

variable_name is a valid C name that has been declared as char type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to getchar function. Since **getchar** is used on the right-handed side of an assignment statement, the character value of getchar is in turn assigned to the variable name on the left.

Example

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

getchar reads one character at a time from the standard input, which is generally the keyboard, unless it is re-directed by the operating system. The **getchar** function reads the character one at a time till it encounters the special character EOF, i.e. End-of-file. **putchar** is the counterpart of **getchar** which is used for displaying one character at a time onto the terminal or console.

getchar reads a character from the standard input device, while **putchar** writes a character to the standard output device.

The putchar takes the following form :

```
putchar (variable_name);
```

where variable_name is a type char variable containing a character. This statement displays the character contained in the variable_name at the terminal.

Example

```
answer = 'Y';
putchar(answer);
```

will display the character Y on the screen. The statement

```
putchar(answer);
```

would cause the cursor on the screen to move to the beginning of the next line.

Example

/* Program illustrates both getchar and putchar */

```
#include <stdio.h>
void main( )
{
    char c;
    printf("Enter one character : ");
    c = getchar( );
    printf("The character you entered is : ");
    putchar(c);
}
```

Output :

```
Enter one character : n
The character you entered is : n
```

String I/O : gets and puts

The function **gets** receives the string from the standard input device, while **puts** outputs the string to the standard output device.

Example

```
/* Program illustrates both gets and puts */
#include <stdio.h>
void main()
{
    char c[80];
    printf("Enter the string : ");
    gets(c);
    printf("The string you entered is : ");
    puts(c);
}
```

Output

```
Enter the String : Naren
The string you entered is : Naren
```

gets

The methods of calling getchar and gets differ, notice that getchar does not take any parameter and returns the value of the input character; which is assigned to the variable.c. However, gets appears in the statement as shown below

```
gets(c);
```

The function gets accepts the name of a string as a parameter, and fills the string with characters that are input from the keyboard, till a newline character is encountered. At the end, the function gets appends a null terminator as must be done for any string, and returns. The newline character is not added to the string.

puts

In the end, a statement,

```
puts(c)
```

will display the string stored in c. This is the same string that was read with gets previously.

Example

C program accepts a given character and display the next character from the ASCII table.

```
#include<stdio.h>
int main(void)
{
    int ch;
    ch='A';
    ch=ch + 1;
    putchar(ch);
    return 0;
}
```

Output : B

Here, the character "A" is assigned to the integer variable "ch". This results in the ASCII equivalent, 41h, of the character "A" being stored in the lower order byte of the integer variable "ch". As 1 is added to the contents of "ch", it becomes

42h, which is the ASCII representation for the character “B”. So when `putchar(ch)` is executed the character displayed on the screen is “B”.

2. Formatted Input/Output Operations

The formatted Input / Output functions can read and write values of all data types, provided the appropriate conversion symbol is used to identify the datatype.

- **scanf** is used for receiving formatted Input and
- **printf** is used for displaying formatted output.

Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set. (ASCII is the American Standard Code for Information Interchange. It is the character set used by almost all current computers, with the notable exception of large IBM mainframes.) Formatted files are human readable and can be typed to the terminal screen or edited with a text editor.

printf() Function

`printf()` is used to display information on the standard output, which is usually the screen. A call to `printf()` contains a string argument and may include additional arguments. For each placeholder – that is, each character sequence beginning with % (but not with %%) – and additional argument is included in the call.

Any material in the first (string) argument other than placeholders is written to the standard output as is. If any escape sequences are contained in the string argument, these are interpreted and executed. For example, if the string argument includes the `\a` escape sequences, an alarm will be sounded during the display.

In addition to the contents of the string argument, any information specified by a placeholder is written to the standard output. Such placeholder information is written in a format specified as part of the placeholder.

In the simplest form, a placeholder will consist of % and a letter indicating a type (such as d, f, or c). This information may be augmented by any combination of the following.

- Flags, which modify the output format.
- A field width, which specifies the number of digits for a whole number, or the number of digits to the right of the decimal point in the output for a floating point value.
- A prefix, which modifies the output type. For example, `h` indicates a short int, and `l` indicates a long form of a type, whether whole number or floating point.

Any modifiers must appear in the order listed. Thus, a placeholder must have the first and the last of the following components, and can have any of the intervening ones in the order shown :

- flags field-width precision prefix type-identifier.

char	Use for printing
d	Integers
u	Unsigned integers
o	Octal integers
x	Hexadecimal integers, using a-f
X	Hexadecimal integers, using A-F
f	Floating point numbers
e	Floating point numbers in exponential format, using e before
E	Floating point numbers in exponential format, using E before
g	Floating point numbers in f or e format
G	Floating point numbers in f or E format
c	Single characters
s	Null-terminated character strings
%	Percent signs

Table : printf conversion characters

scanf() Function

scanf() is used to read information from the standard input, which is generally the keyboard. Recall that a call to **scanf()** contains a string argument and may include additional arguments. For each placeholder – that is, each character sequence beginning with % (but not with %%) – and additional argument is included in the call. Recall that additional arguments to **scanf()** must be pointers (addresses).

When **scanf()** is called, the input is processed and is matched against the string argument. As long as the processing produces matches, the processing of the input line continues.

- If a placeholder is encountered in the string argument to **scanf()**, a value of the specified type as the next element in the input constitutes a match.
- If one or more whitespace characters are encountered in the string argument, any sequence of one or more whitespace characters in the input constitute a match.
- If any other character is encountered in the string argument to **scanf()**, the occurrence of this same character in the input constitutes a match.

The contents of the string argument are sought in the input. As soon as there is a discrepancy between the string argument and the input, the function fails and terminates. Discrepancies occur because expected input is missing or because actual input differs from expected input.

If the string argument contains any text other than that for placeholders, this text must also be compared to the input. If the string argument contains one or more whitespace characters, the input is processed until the first non-whitespace character is encountered. Processing stops just before this character. This means that you can include whitespace in your string argument without having to include the same amount of whitespace in the input. (The exception to this is if you are expecting to read character input).

If the text is neither whitespace characters nor a %, then the input is processed and matching text is expected. If it is not found, the function fails and terminates.

Consider this program :

```
/* Program to illustrate scanf( ) function */
#include <stdio.h>
main ( )
{
    int i1, i2, i3 ;
    /* input required: <val> "then" <val> <val>
       spaces between vals and strings are not fixed */
    scanf ("%d the %d %d",
           &i1, &i2, &i3) ;
    printf ("i1 == %5d; i2 == %5d; i3 == %5d\n",
           i1, i2, i3;
}
```

Output :

```
12    the    16    35
11        the 14    23
11 14 23
```

The first and second of the following lines are valid input; the third is not, since, "the" is missing. Notice that when the string argument contains a whitespace character, any number of whitespace characters will be skipped.

The *conversion-string* can contain three types of *directives* :

➤ Regular	This is text that must be matched character by character with the
characters	input. Such entries are rarely used for interactive programs, but can be handy when working with formatted input files.
➤ White-space	A blank, tab, or other white-space character will match any
characters	amount (including none) of any white-space. (So a single space will match any string of white-space, including newlines.) Note that it is legal for this to match no input at all (if there isn't a blank or tab, it is ok).
➤ Conversion	Similar to printf conversion specifiers but just different enough to
Specifiers	cause many errors. They all begin with a percent and end with a letter indicating the type of conversion. In between can be some special conversion controls, including the <i>length</i> . Unlike printf, failing to use the exact type and length for the conversion will result in unpredictable errors. Since few compilers will check the conversion-string for argument mismatches, the result is a runtime (logic) error that can be

hard to find. These conversion specifiers match a string of characters in the input, convert to the specified type (and length), and store the result in the RAM address provided by the corresponding argument.

char	Action
d	The value to be read is expressed in decimal notation; the corresponding argument is a pointer to an int unless the <i>l</i> or <i>h</i> modifiers is used, in which case the argument is a pointer to a long or short int, respectively.
u	The value to be read is an integer and the corresponding argument is a pointer to an unsigned int.
o	The value to be read is expressed in octal notation; the corresponding argument is a pointer to an int, unless an <i>l</i> or <i>h</i> precedes the x, in which case the argument is a pointer to a long or a short respectively.
x	The value to be read is expressed in hexadecimal notation; the corresponding argument is a pointer to an int, unless an <i>l</i> or an <i>h</i> modifies the x.
e, f, g	The value to be read is expressed in floating point notation; the value may be optionally preceded by a sign and may optionally be expressed in exponential notation; the corresponding argument is a pointer to a float, unless an <i>l</i> modifier is used, in which case it is a pointer to a double.
c	The value to be read is a single character; the next character that appears on the input is read, even if it is a space, tab, newline or formfeed character; the corresponding argument is a pointer to a character; an optional count before the c specifies the number of characters to read.
s	The value to be read is a sequence of characters; the sequence begins with the first nonwhitespace character and is terminated by the first whitespace character; the corresponding argument is a pointer to a character array, which must contain enough character to contain the characters that are read plus the null character that is automatically added to the end of the string; if a number precedes the s, then the specified number of characters is read, unless a whitespace character is encountered first.
[. . .]	Characters enclosed within brackets indicate that a character string is to be read, as in %s; the characters within the brackets indicate the permissible characters in the string; if any character other than that specified in the brackets is encountered, then the string will be terminated.

Table : Scanf Conversion Characters

Example : Program adds two integer numbers and print the input numbers and result.

```
#include <stdio.h>
int main()
{
    int a,b,c;
    printf("\nThe first number is ");
    scanf("%d",&a);
    printf("\nThe second number is ");
    scanf("%d",&b);
    c=a+b;
```

```
printf("The answer is %d \n",c);  
return 0;  
}
```

Output :

The first number is 6
The second number is 7
The answer is 13

Example : Program prints formatted floating-point number

```
#include <stdio.h>  
int main()  
{  
    printf("%f\n",1234.56);  
    printf("%e\n",1234.56);  
    printf("%4.f\n",1234.56);  
    printf("%3.1f\n",1234.56);  
    printf("%-10.3f\n",1234.56);  
    printf("%10.3f\n",1234.56);  
    printf("%10.3e\n",1234.56);  
    return 0;  
}
```

Output : /1234.560000/
/1.234560e+03/
/1235/
/1234.6/
/1234.560 /
/ 1234.560/
/ 1.235e+03/

Example

Write a program that prints the next character for the corresponding three characters given to the program.

```
#include <stdio.h>  
int main()  
{  
    char a,b,c;  
    scanf("%c%c%c", &a, &b, &c);  
    a++;
```

```

b++;
c++;
printf("a=%c b=%c c=%c",a,b,c);
return 0;
}

```

Input : PQR

Output : a=Q b=R c=S

2.2 ESCAPE SEQUENCES

Q3. Explain escape sequences in C - languages ?

Ans :

(Nov.-20, June-18)

Certain nonprinting characters, as well as the backslash (\) and the apostrophe ('), can be expressed in terms of escape sequence. An escape sequence always begins with a backslash and is followed by one or more special characters. An escape sequence is a means of giving special output instructions. For example, a line feed (LF), which is referred to as a new line in C can be represented as \n. Such escape sequence always represents single character, even though they are written in terms of two or more characters.

The commonly used escape sequence are listed below :

Code	Description
\a	bell (beep)
\b	backspace
\t	horizontal tab
\v	vertical tab
\n	new line
\f	form feed
\r	carriage return
\"	double quotation mark
\'	apostrophe
\?	question mark
\\	backslash
\0	null

2.3 CONTROL STATEMENTS

Q4. Explain Control Statements with suitable examples?

Ans :

A C program is a set of statements which are normally executed sequentially in the order in which they appear. We have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. C language possesses such decision-making capabilities by supporting the following statements :

1. if statement
2. switch statement
3. conditional operator statement
4. goto statement

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

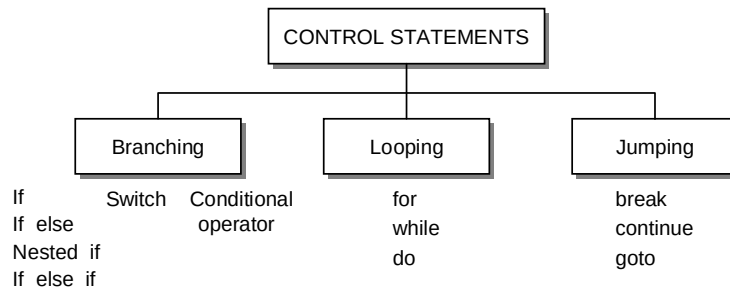


Fig. : Broad Classification of Control Statements

2.3.1 Selection Statements / Decision Making Statements / Branching Statements

Q5. Explain about Selection Statements.

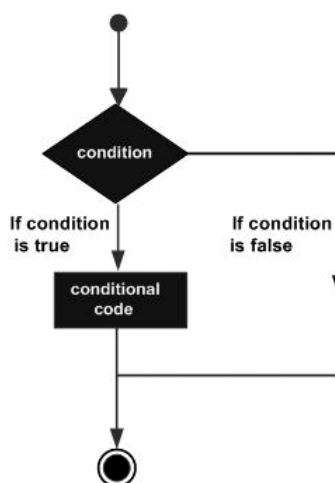
Ans :

(Dec.-18)

if

Selection statements are also called decision making statements. These are used when a condition arises in a statement. Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision making structure found in most of the programming languages -



C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

The various selection statements used are as :

- 1) If
- 2) If Else
- 3) Nested If
- 4) If Else If
- 5) Switch
- 6) Conditional Operator
- 7) Comma Operator

2.3.1.1 if control statement

Q6. Explain about if statement

Ans :

if

The 'if' control statement is a conditional statement which checks the condition given as an argument to the statement and if evaluated to true continues the execution of the statements that are existing within the statement.

Syntax

```
if (<condition>)  
{  
    Statement 1;  
    Statement 2;  
    ---  
    ---  
    Statement n;  
}
```

Example

/* Program to explain simple if statement */

```
#include<stdio.h>  
main( )  
{  
    int s;  
    printf("Enter score \n\n");  
    scanf("%d",&s);  
    if(s>100) /*if statement in brackets a condition is there if the condition true,  
              the statement after if will be executed. In s>100 is relational operator*/
```

```

        {                               /* if operating */
            printf("score is significant \n"); /* statements for if */
            printf("india won the match \n");
        }                               /* if closing */
    }

```

Output

```

Enter score
123
score is significant
india won the match

```

2.3.1.2 if ... else Control Statement**Q7. Explain about if ... else Control Statement.****Ans :****(Nov.-20, Imp.)**

This control statement works exactly same as the *if* statement that we have discussed above but, here we provide the alternative statements to be executed if the given condition is evaluated to false that is the *else* part.

Syntax

```

if (<condition>)
{
    Statement 1;
    ---
    Statement n;
}
else
{
    Statement 1;
    ---
    Statement n;
}

```

Example

```

#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 100;
    /* check the boolean condition */
    if( a < 20 ) {
        /* if condition is true then print the following */

```



```
        printf("a is less than 20\n" );
    }
    else {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

Output :

```
a is not less than 20;
value of a is : 100
```

2.3.1.3 Nested if...else, if ...else if Statements

Q8. Explain about Nested if...else Statement.

Ans :

(Nov.-20)

A if structure can have one more if (if...else) structure as a part of statements and even an else part of an if...else structure can also have again an if...else structure. This sort of building the if...else structure is called Nested if...else structure.

Syntax

```
if (<condition>)
{
    statements to be done;
    if (<condition>)
    {
        statements to be done;
    }
    else
    {
        statements to be done;
    }
}
else
{
    statements to be done;
    if (<condition>)
    {
        statements to be done;
    }
}
```

```
        else
        {
            statements to be done;
        }
    }
```

Example

Program prints the largest of the three numbers using nested if ...else statements

main()

```
{
    float A, B, C;
    printf("Enter three values \n");
    scanf("%f %f %f", &A, &B, &C);
    printf("\n Largest value is ");
    if (A > B)
    {
        if (A > C)
            printf("%f\n", A);
        else
            printf("%f\n", C);
    }
    else
    {
        if (C > B)
            printf("%f\n", C);
        else
            printf("%f\n", B);
    }
}
```

Output

```
Enter three values
352  44983  9483
Largest value is  44983.000000
```

Q9. Explain about if ... else if Statement.

Ans :

(June-17)

if...else if structure is the other way of representing an if...else...if structure, i.e. instead of using 'if' statement again in the else part of the if...else structure we can use a single statement as 'else if'. This statement works exactly the same as 'if...else...if' structure.

Syntax

```

if (condition)
{
    statements to be done;
}
else if
{
    statements to be done;
}

```

2.3.1.4 Switch Statement**Q10. Explain about Switch Statement.**

Ans : (Dec.-18, June-17)

The **switch** statement selects a particular statement from a group of statements. The selection depends upon the current value of the expression where the expression results in an integer value, character value or any other data type.

The general form of the switch statement is

```
switch (expression) statement
```

where expression results in an integer value. Note that expression may also be of type char, since individual characters have equivalent integer values.

The embedded statement is generally a compound statement that specifies alternative courses if action. Each alternative is expressed as a group of one or more individual statements within the overall embedded *statement*.

In general terms, each group of statements is written as

```

case expression :
    statement 1
    statement 2
    ... ..
    statement n

```

or, when multiple case labels are required,

```

case expression 1 :
case expression 2 :
    ... ..

```

```

case expression m :
    statement 1
    statement 2
    ... ..
    statement n

```

where expression 1, expression 2, . . . , expression m represent constant, integer-valued expressions. Usually, each of these expressions will be written as either an integer constant or a character constant. Each individual statement following the case labels may be either simple or complex.

When the switch statement is executed, the expression is evaluated and control is transferred directly to the group of statements whose case-label value matches the value of the expression.

Example

```

switch (choice = getchar( ))
{
    case 'r'
    case 'R' :
        printf("Red");
        break;

    case 'w':
    case 'W' :
        printf("White");
        break;

    case 'b' :
    case 'B':
        printf("Blue");

}

```

Red will be displayed if choice represents either r or R, White will be displayed if choice represents either w or W, and Blue will be displayed if choice represents either b or B.

2.3.1.5 Comma Operator**Q11. Explain about Comma Operator***Ans :***Comma Operator**

The comma operator (,) allows two or more different expressions to appear when only one expression is expected. The expressions are separated by the comma operator. The comma operator takes its arguments and evaluates them from left to right and returns the value of the rightmost expression.

Consider the following statement :

```
i = (j = 10, k = 5, j + k);
```

Here, the operator first evaluates 'j = 10' which assigns 10 to the variable j. Then the next expression 'k = 5' is evaluated which assigns 5 to k. The third expression is evaluated and its value returned as the operator's result. Thus, i is assigned 15.

2.3.1.6 Conditional Operator**Q12. Explain about Conditional Operator.***Ans :* (Dec.-16)**Conditional Operator**

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and : and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use the conditional operator is as follows :

```
Exp1 ? Exp2 : Exp3;
```

where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.

- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

For example, the segment

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

can be written as

```
flag = (x < 0) ? 0 : 1;
```

Consider the evaluation of the following function :

```
y = 1.5x + 3 for x ≤ 2
```

```
y = 2x + 5 for x > 2
```

This can be evaluated using the conditional operator as follows :

```
y = (x > 2) ? (2 * x + 5) : (1.5 * x + 3);
```

The conditional operator may be nested for evaluating more complex assignment decisions. For example, consider the weekly salary of a salesgirl who is selling some domestic products. If x is the number of products sold in a week, her weekly salary is given by

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as

```
salary = (x != 40) ? ((x < 40) > (4*x+100) : (4.5*x+150) : 300;
```

The same can be evaluated using if...else statements as follows :

```
if (x <= 40)
    if (x < 40)
        salary = 4 * x + 100;
    else
```

```

    salary = 300 ;
else
    salary = 4.5 * x+150 ;

```

When the conditional operator is used, the code becomes more concise and more efficient.

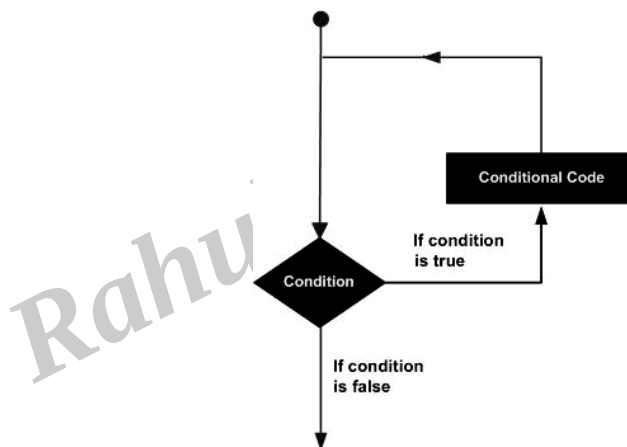
2.4 LOOPING STATEMENTS

Q13. What are loops ? Explain the loop control statements in C language ?

Ans :

A portion of program that is executed repeatedly is called a loop. A loop allows one to execute a statement or block of statements repeatedly. There are mainly two types of iterations or loops – *unbounded iteration* or *unbounded loop* and *bounded iteration* or *bounded loop*. In bounded iteration, repetition is implemented by constructs that allow a determinate number of iterations. That is, bounded loops should be used when we know, ahead of time, how many times we need to loop. C provides for construct as bounded loop.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages -



The C programming language contains three different program statements for program looping. They are :

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

2.4.1 while loop

Q14. Explain about while loop.

Ans :

(Dec.-19, June-18, Dec.-16)

If you want to perform an action repeatedly to the different values of a variable, the while loops is ideally used. For example if you want to calculate simple interest for different values of principle, period and rate of interest. We can make use of while loop.

Syntax

```
while (condition)
{
    statement to done;
}
```

Example

```
#include<stdio.h>
main( )
{
    int p, n;
    float r, s;
    int cnt = 1;
    while (cnt<=2)
    {
        printf("Enter the values of p, n & r");
        scanf("%d%d%f; & p, &n, &r);
        s = p*n*r/100;
        printf("Simple interest = Rs. %f",s);
        cnt = cnt+1;
    }
}
```

Output :

```
Enter the values of p, n & r
4
1000
10.5
Simple interest = Rs. 420
Enter the values of p, n & r
6
5023
11
Simple interest = Rs. 3315.179932
```

2.4.2 do-while loop**Q15. Explain do-while loop.**

Ans :

(Dec.-19, June-18, Dec.-16)

do-while loop

The structure of **do-while** loop is similar to while loop. The difference is that in case of **do-while** loop the expression is evaluated after the body the loop is executed. In case of while loop the expression is evaluated before executing body of loop.

The general form of **do-while** statement is

```
do
    statement;
while (expression);
```

where statement is a single statement or compound statement. In contrast to while loop statement (body of loop), do-while loop is executed one or more times.

In do-while loop, first the statement is executed and then the expression is tested. If the expression evaluates to false, the loop terminates and the execution control transfer to the next statement that follows it.

The following program illustrates the use of do-while loop.

```
# include <stdio.h>
main( )
{
    int counter = 1;
    /* loop control variable */
    do
    {
        printf("Now the counter value is %d\n", counter);
        counter++;
    }
    while(counter <= 10);
    printf("End of the loop");
}
```

Here, the body of loop is repeated 10 times.

In case of *do-while* loop, the statement is executed at least once, whereas in case of while loop, the statement may not execute at all if the expression results false for the first time itself.

Program to add first n even numbers using do-while loop.

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

```
do
{
    sum = sum + i;
    i++;
    i++;
}
while (i < n);
printf("Sum of first %d even number is %d\n", n, sum);
}
```

2.4.3 for Loop

Q16. Explain for Loop.

Ans :

(Dec.-19, June-18, June-17, Dec.-16)

for Statement

The for loop is most common in major programming languages. However, the for loop in C language is very flexible and very powerful. The for loop is used to repeat the execution of a statement for fixed number of times.

The general form of the for loop is

```
for (initialization; test-condition; increment)
{
    Body of the loop
}
```

Execution process of the for statement is as follows :

- Initialization of the control variables is done first, using assignment statements such as $i = 1$ and **count** = 0. The variables i and **count** are known as loop-control variables.
- The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as $i < 10$ that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the *for* statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as $i = i + 1$ and the new value of the control variable is again tested to see whether it satisfies the loop condition.

If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Consider the following segment of a program:

```
for (x =0; x <= 9; x = x + 1)
```



```

{
    printf("%d", x);
}

printf("\n");

```

This for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons.

The for statement allows for *negative increments*. For example, the loop shown above can be written as follows :

```

for (x =0; x <= 9; x = x - 1)

    printf("%d", x);

printf("\n");

```

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Program to find the sum of 10 given numbers using for statement.

```

#include <stdio.h>

main( )
{
    int number, i, sum = 0;

    printf("Enter values for 10 numbers \n");

    for( i = 1, i < 10; i++ )
    {
        scanf("%d\n", &number);

        sum = sum + number;
    }

    printf("Sum if %d\n", sum);
}

```

C programming allows to use one loop inside another loop.

Syntax :

The syntax for a nested **for** loop statement in C is as follows :

```

for ( init; condition; increment ) {
    for ( init; condition; increment ) {
        statement(s);
    }
    statement(s);
}

```

The syntax for a nested **while** loop statement in C programming language is as follows:

```

while(condition) {
    while(condition) {
        statement(s);
    }
    statement(s);
}

```

The syntax for a nested **do...while** loop statement in C programming language is as follows:

```

do {
    statement(s);
    do {
        statement(s);
    } while( condition );
} while( condition );

```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 50 :

```

#include <stdio.h>

int main () {

    /* local variable definition */

    int i, j;

```

```
for(i = 2; i<50; i++) {  
    for(j = 2; j <= (i/j); j++)  
        if(!(i%j)) break;    // if factor found, not prime  
    if(j > (i/j)) printf("%d is prime\n", i);  
}  
return 0;  
}
```

Output :

2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

2.5 SPECIAL CONTROL STATEMENTS (GOTO, BREAK, CONTINUE, RETURN, EXIT)

Q17. Explain Special statement in loops of C - language.

Ans :

(Nov.-20)

Loop performs a set of statements, till the condition becomes false. The number of times a loop is to be repeated is decided in advance and a condition is set. But sometimes it becomes necessary to skip certain part of the loop or terminate the loop if a particular condition is true then jump statements come to action.

Jump statements consist of the following types of statements :

1. The break statement
2. The continue statement

3. The return statement
4. The goto statement
5. Exist statement

1. break Statement

The break statement is used to terminated loops or to exit from a switch. It can be used within a for while, do-while, or switch statement.

The break statement is written simply as

```
break;
```

without any embedded expression or statements

The break statement causes a transfer a control out of the entire switch statement, to the first statement following the switch statement.

```
switch (choice = toupper(getchar( )))  
{  
case 'R' :  
    printf( "RED" );  
    break;  
case 'W' :  
    printf( "WHITE" );  
    break;  
case 'B' :  
    printf( "BLUE" );  
    break;  
default :  
    printf( "ERROR" );  
    break;  
}
```

Each group of statements ends with a break statement, in order to transfer control out of the switch statement. The break statement is required within each of the first three groups, in order to prevent the succeeding groups of statements from executing. The last group does not require a break statement, since control will automatically be transferred out of the switch statement after the last group has been executed.

2. Continue Statement

The *continue* statement is used to *bypass* the remainder of the current pass through a loop. The loop does *not* terminate when a *continue* statement is encountered. The remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.

The continue statement can be included within a *while*, a *do-while* or a *for* statement. It is written simply as

continue;

without any embedded statements or expressions.

Example

Program demonstrates the continue statement using do-while loop

```
do {
    scanf( "%f", &x );
    if ( x < 0 ) {
        printf("ERROR - Negative value for X");
        continue;
    }
    /* Process the nonnegative value of x */
    ... ..
} while ( x <= 100);
```

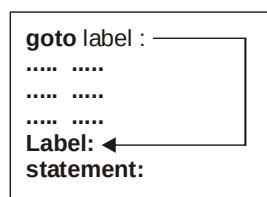
Using for loop

```
for (count = 1; x <= 100; ++count) {
    scanf( "%f", &x );
    if ( x < 0 ) {
        printf("ERROR - Negative value for X");
        continue;
    }
    /* Process the nonnegative value of x */
    ... ..
}
```

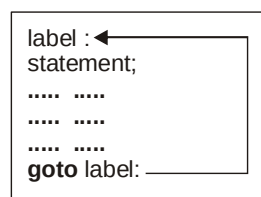
3. goto Statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the *goto* might be desirable.

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



Backward jump

The *label* can be anywhere in the program either before or after the **goto** label; statement. During running of a program when a statement like

```
goto begin ;
```

is met, the flow of control will jump to the statement immediately following the label begin:. This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the *label*: is before the statement `goto label`: a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump*. On the other hand, if the *label*: is placed after the `goto label`; some statement will be skipped and the jump is known as a *forward jump*.

A goto is often used at the end of a program to direct the control to go to the input statement, to read further data.

Consider the following example :

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

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two goto statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.

Due to the unconditional goto statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite loop*.

Program evaluates the square root for five numbers using of the goto statement

```
#include <math.h>
main( )
{
    double x, y;
    int count ;
    count = 1 ;
    printf("Enter FIVE real values in a LINE \n");
read:
    scanf("%1f", &x) ;
```

```
printf("\n") ;
if (x < 0)
    printf("Value – %d is negative\n", count) ;
else
{
    y = sqrt(x) ;
    printf("1ft %1ft\n", x, y) ;
}
count = count + 1;
if (count <= 5)
goto read ;
printf("\nEnd of computation");
}
```

4. return Statement

The return statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A return statement can also return a value to the calling function.

Syntax

jump-statement:

return **expression** opt ;

The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined. The expression, if present, is converted to the type returned by the function. If the function was declared with return type void, a return statement containing an expression generates a warning and the expression is not evaluated.

If no return statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. In this case, the return value of the called function is undefined. If a return value is not required, declare the function to have void return type; otherwise, the default return type is int.

Many programmers use parentheses to enclose the *expression* argument of the return statement. However, C does not require the parentheses.

Example below demonstrates the return statement :

```
void draw( int l, long L );
long sq( int s );
int main( )
{
    long y; int x;
    y = sq( x );
}
```

```
        draw( x, y );
        return( );
    }
    long sq( int s )
    {
        return( s * s );
    }
    void draw( int I, long L )
    {
        /* Statements defining the draw function here */
        return;
    }
```

Here, the main function calls two functions : *sq* and *draw*. The *sq* function returns the value of $x * x$ to main, where the return value is assigned to *y*. The *draw* function is declared as a void function and does not return a value. An attempt to assign the return value of *draw* would cause a diagnostic message to be issued.

5. exit - statement

exit is not a jump statement. It is used to terminate the program or directly exit from the program without following the intermediate statements. It is a library function included in *stdlib.h* and *process.h*. *exit* is used as *exit()*. Whenever the *exit* function is encountered, the program execution is brought to an end. It can be included at any position in the program body. Its general form is given below.

Syntax : *exit()*

Example :

```
for(i = 0; i < 100; i++)
{
    scanf("%d", &a);
    if (a >= 100)
    {
        printf("Greater");
    }
    else
    {
        exit();
    }
}
```

Here, if the input integer variable value becomes less than 100, then the *exit* function is encountered and the whole program execution is terminated i.e. brought to an end.

2.6 ARRAYS & STRINGS**Q18. Define arrays ? What is the need of arrays ?****Ans :****(June-19, Dec.-18)**

C programming language allows working with collection of data elements of same datatype, called as an array. The elements of an array are stored sequentially in memory. In C, string are stored as an array of characters, terminated with a null character.

In C we can define variable called grades, which represents not a single value of grade but a entire set of grades. Each element of the set can then be referenced by means of a number called as index number or subscript.

Need for Arrays

The need for arrays comes from the problem of representing large amounts of data in the program. It might often be required to store a number of items, say, for processing. The items may be all related in the fact that they all represent a particular theme. However, to declare n variables for storage of the n items is tedious and a non-elegant approach. As programmers, we represent such sequences by arrays.

In C, an array is an aggregate variable consisting of a collection of the same type. The array identifier is the name given to the entire collection; individual items are identified in terms of their location in the array. The elements in an array are stored in consecutive memory locations. Arrays in C can be broadly classified into two groups – single or one-dimensional and multi-dimensional.

2.6.1 One Dimensional Arrays**Q19. Explain the declaration and Accessing of one dimensional array ?****Ans :****(June-19, Dec.-18, Imp.)**

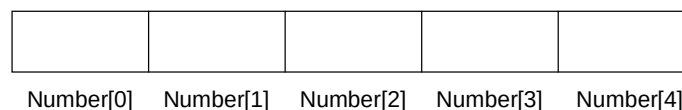
A list of item can be given one variable name using only one subscript and such a variable is called a one dimensional array.

Example

If we want to store a set of five numbers by an array variable number, then it will be accomplished in following way :

```
int number[s];
```

This declaration will reserve five contiguous memory locations as shown below.

**Fig.: Structure of a single dimensional array**

As 'C' performs no bounds checking, therefore, care should be taken to ensure that the array indices are within the declared limits.

Declaration of Arrays

All variables in C must be declared before use. Hence, we must first declare the array. It is done at the beginning of the function `main()` with the statement we must declare the array. It is done at the beginning of the function `main()` with the statement

```
int age [5];
```

This statement states that `age` is an array of 5 integers. Note the use of square braces after the *array name* (i.e., `age`) enclosing the integer 5 that gives the number of elements in the array. This statement reserves enough memory for `age` to hold 5 integers. Once the declaration is made, `age[0]` refers to the first location in the array, `age[1]` refers to the second location and so on. Finally, `age[4]` refers to the last integer in the array. In C, array indices always begin with zero. Figure below shows the declaration of an array.

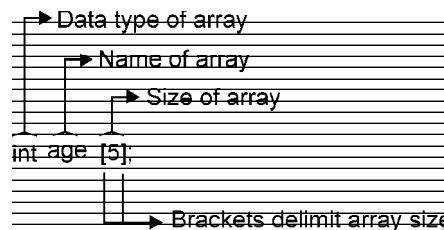


Fig. : Array definition

```
# include <stdio.h>
void main()
{
    int ages [ 5 ]; /*Array of 5 ages, declaration for an array */
    int i, n;
    float sum = 0;
    printf("Number of persons: ");
    scanf("%d", &n);
    if (n <= 0 || n > 5)
        printf("Invalid number of persons entered.\n");
    else
    {
        for(i = 0; i<n; i++)
        {
            printf("Age:");
            scanf("%d", &age[ i ] );
            sum + = age [ i ];
        }
    }
}
```

```

        printf("The age input are: \n");
        for ( i = 0; < n; i++)
            printf("%d\n", age [ i ] ) ;
        printf("The average is: %f\n", sum/n) ;
    }                /* end else */
}                /* end main */

```

Output :

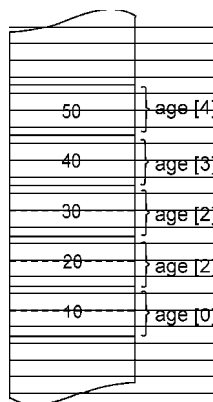
```

Number of persons : 5
Age : 10
Age : 20
Age : 30
Age : 40
Age : 50
The ages input are :
10
20
30
40
50
The average is : 30.000000

```

Accessing Array Elements

To access a particular element in an array, specify the array name, followed by square braces enclosing an integer, which is called the *array index*. The array index indicates the particular element of the array which we want to access. The numbering of elements starts from zero (i.e., age[0] is the age of first person). Figure below the elements of an array in memory. It is assumed that each element of the array (i.e., each integer) occupies two bytes.

**Fig.: Elements of the array age**

Observe the loop used to input the elements. It is produced here four convenience.

```
for(i = 0; i < n; i++)
{
    printf("Age : ");
    scanf("%d", &age [ i ] );
    sum += age [ i ] ;
}
```

The variable *i* varies from 0 to *n*-1 (0 to 4 in the above sample run). The function *scanf* is called to input an integer value. The address of the *i*th location is passed to *scanf*. This makes *scanf* store the integer input into successive locations each time the loop is executed. Note that

&age [*i*]

in the *scanf* statement refers to the memory location of the integer at the *i*th position.

The statement

```
sum += age [ i ] ;
```

adds the value of the *i*th integer to *sum*.

The rest of the program illustrates the use of the array variable in various situations. The first for loop is used to add to the variable *sum*, and in the second, to print out the values of *age* that are input.

An element such as *age*[*i*] can be used just like any other ordinary integer. Statements such as *age*[*i*] ++; (to increment the value of the *i*th integer in the array *age*) and *age*[*i*] = 11; are perfectly valid.

2.6.2 Initialization of One Dimensional Array

Q20. Explain how to initialize one dimensional array ?

Ans : (June-17)

Arrays can be initialized once they are declared. The declaration of an array can be done as follows.

```
int a[ 5 ] = {13, 34, 2, 6, 234};
```

Here, *a* is an array of 5 integers. The initial value of the five integers are specified in order, between the curly braces, separated by commas. The above declaration causes *a*[0] to be initialized to 13, *a*[1] to 34 and so on till *a*[4], which is initialized to 234. A semicolon always follows the closing brace.

The array size may be omitted during declaration, as shown below.

```
int a [ ] = {13, 34, 2, 6, 234};
```

This declaration is equivalent to the first one. In this case, the subscript is assumed to be equal to the number of elements in the array (5 in this case). Figure below shows the initialization of an array.

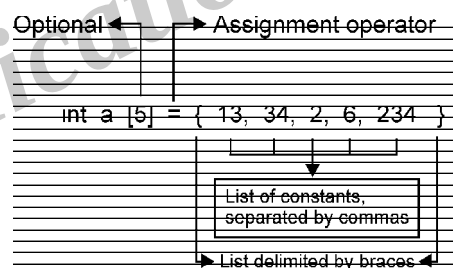


Fig. : Array Initialization

C does *not* check the validity of the subscript when you use arrays. For example, take a look at the program below :

```
void main( )
{
    int a[40] ;
    a[50] = 11;
    a[50]++;
}
```

The program declares *a* as an array of 40 integers, and then modifies the 51st element! When you compile the above program, you do not get any error. The executable code is produced. When

it is run, the results are unpredictable. Hence, we must be very careful while using array subscripts. A wrong subscript in a large program can cause it to behave erratically, demanding a lot of debugging time.

Similarly to an array of integers, we can have an array of any data type. For example, the declaration for an array of 50 floating-point numbers is :

```
float f [50];
```

The individual floating-point numbers in this array are accessed as before, i.e., f[0] is the first floating-point number and f[49] the last.

An example to find out the largest and smallest element in an array is illustrated below :

```
/* Program to find largest and smallest element in the vector */
```

```
# include <stdio.h>
main( )
{
    int i, n;
    float a[50], large, small;
    printf ("Size of vector ? ");
    scanf ("%d", &n) ;
    printf ("\nVector elements ?\n");
    for (i = 0; i < n; i++)
        scanf ("%f", &a[ i ] ) ;
    /* Initialization */
    large = a[0] ;
    small = a[0] ;
    /* largest and smallest elements */
    for (i = 1; i < n ; i++)
    {
        if (a[i] > large)
            large = a[i] ;
        else if (a[ i ] < small)
            small = a [ i ];
    }
    printf ("\nLargest element in vector is %8.2f\n", large) ;
    printf ("\nSmallest element in vector is %8.2f\n", small) ;
}
```

Output :

```
Size of vector ? 7
Vector elements ?
34.00 -9.00 12.00 10.00 -6.00 0.00 36.00
```

Largest element in the vector is 36.00

Smallest element in vector is -9.00

The initialization of arrays in C suffers two drawbacks

1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method to initialize large number of elements.

2.6.3 Two/Multidimensional Arrays

Q21. What is Multidimensional Arrays ? Explain the initialization and accessing of Multidimensional Arrays.

Ans : (Nov.-20, Dec.-19, June-19, Dec.-18, Dec.-17, Imp.)

Often there is a need to store and manipulate two dimensional data structure such as matrices & tables. Here the array has two subscripts. One subscript denotes the row & the other the column.

The declaration of two dimension arrays is as follows :

```
data_type array_name[row_size][column_size];
int m[10][20];
```

Here m is declared as a matrix having 10 rows(numbered from 0 to 9) and 20 columns (numbered 0 through 19). The first element of the matrix is $m[0][0]$ and the last row last column is $m[9][19]$.

Elements of Multidimensional Arrays

A 2 dimensional array marks [4][3] is shown below figure. The first element is given by marks [0][0] contains 35.5 & second element is marks [0][1] and contains 40.5 and so on.

marks [0][0] 35.5	Marks [0][1] 40.5	Marks [0][2] 45.5
marks [1][0] 50.5	Marks [1][1] 55.5	Marks [1][2] 60.5
marks [2][0]	Marks [2][1]	Marks [2][2]
marks [3][0]	Marks [3][1]	Marks [3][2]

Initialization of Multidimensional Arrays

Initialization of a two-dimensional array during declaration is done by specifying the elements in *row major* order (i.e., with the elements of the first row in a sequence, followed by those of the second, and so on). An example is given below :

```
int two_dim_int [3] [4] =
{
    { 1, 2, 3, 4 },
    { 2, 3, 4, 5 },
```

```

    { 5, 6, 7, 8 }
};

```

The variable `two_dim_int` is declared as a two dimensional integer array with certain initial values. The first subscript can be omitted. For eg, the declaration below wherein the number of rows(3) is not explicitly specified, is equivalent to the previous one.

```

int two_dim_int [ ] [4] =
{
    { 1, 2, 3, 4 },
    { 2, 3, 4, 5 },
    { 5, 6, 7, 8 }
};

```

Also, the inner braces can be omitted, giving the numbers in one continuous sequence. For example,

```

int two_dim_int[ ][4] = { 1, 2, 3, 4, 2, 3, 4, 5, 5, 6, 7, 8 };

```

has the same effect as the previous example, but is not as readable as before.

Unsize Array Initializations

If unsize arrays are declared, the C compiler automatically creates an array big enough to hold all the initializers. This is called an unsize array. The following are examples of declarations with initialization.

(a) `char e1[]="read error\n";`

(b) `char e2[]="write error\n";`

(c) `int sgrs[ ][2] =`

```

{
    1,1,
    2,4,
    3,9,
    4,16,
};

```

Passing Arrays to Functions

Arrays can be passed as arguments to functions. Consider the following example:

/* Program illustrates passing arrays to functions */

```

# include <stdio.h>
const int Rollno = 4;
const int Subject = 3 ;
/* function declaration */
void display (float marks [Rollno] [Subjects]) ;

```

```

main ( )
{
    float marks[Rollno] [Subjects] =
        /* array initialization */
        { { 35.5, 40.5, 45.5 },
          { 50.5, 55.5, 60.5 },
          { 65.0, 70.0, 75.5 },
          { 80.0, 85.0, 90.0 }
        };
    /* Function call with array as an argument */
    display (marks) ;
} /* end main */

/* Function definition to display elements */
void display (float Studentmarks[Rollno][Subjects] )
{
    int r, s ;
    for (r = 0; r<Rollno; r++)
    {
        printf("\n Rollno %d");
        for (s=0; s<Subjects; s++)
            printf("%10.2f", Studentmarks[ r ] [ s ] ) ;
    } /* end for */
} /* end display */

```

The function declaration in the above program,

```
void display(float marks[Rollno] [Subjects]);
```

can also be written as,

```
void display(float marks[ ] [Subjects[ ] ] ;
```

The function display visualizes marks of the array, where the number of columns is Subjects. While calling the function, the name of the array is sufficient as an argument. Hence, the function call in main is of the form :

```
display(marks) ; /* function call */
```

The name marks represents the memory address of the array and is similar to using a reference argument. Elements of an array are not copied, but the function works with a different name, on the original array itself. Hence, any modifications done to the array in the function are reflected in the array of the caller too. Passing by reference is efficient, as arrays can be very large and copying the entire array into a function consumes time and memory.

The function definition,

```
void display(float StudentMarks [Rollno] [Subject])
```

uses the name of the array, data type and their sizes. Inside the function the elements of the array are accessed by :

```
Studentmarks[ r ][ s ] ;
```

/* Program to add two matrices (Multidimensional arrays) */

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

```
main( )
```

```
{
```

```
int a[10] [10], b[10] [10], c[10] [10], i, j, m, n, p, q;
```

```
printf("Input row and column of A matrix \n") ;
```

```
scanf("%d %d", &n, &m) ;
```

```
printf("Input row and column of B matrix \n") ;
```

```
scanf("%d %d", *p, &q) ;
```

```
/* Checks if matrices can be added */
```

```
if ( (n == p) && (m == q) )
```

```
{
```

```
printf("Matrices can be added\n") ;
```

```
printf("Input A-matrix\n") ;
```

```
for(i = 0; i < n; ++i)
```

```
for(j = 0; j < m; ++j)
```

```
scanf("%d", &a[ i ][ j ] ) ;
```

```
printf("Input B-matrix\n") ;
```

```
for (i = 0; i < n; ++i)
```

```
for (j = 0; j < m; ++j )
```

```
scanf("%d", &b[ i ][ j ] ) ;
```

```
/* Addition of two matrices */
```

```
for (i = 0; i < n; ++i)
```

```
for( j = 0; j < m; ++j)
```

```
c[ i ][ j ] = a[ i ][ j ] + b[ i ][ j ];
```

```
printf("Sum of A and B matrices : \n") ;
```

```
for (i = 0; i < n; ++i)
```

```
{
```

```
for ( j = 0; j < m; ++j)
```

```
printf ("%5d", c[ i ][ j ] ) ;
```

```
printf ("\n") ;
```

```
}
```



```
    } /* endif */  
    else  
        printf("Matrices cannot be added\n") ;  
    } /* main */
```

Output 1 :

```
Input row and column of A matrix  
3 3  
Input row and column of B matrix  
1 2  
Matrices cannot be added
```

Output 2 :

```
Input row and column of A matrix  
3 3  
Input row and column of B matrix  
3 3  
Matrices can be added  
Input A-matrix  
1 2 3  
4 5 6  
7 8 9  
Input B-matrix  
1 2 3  
4 5 6  
7 8 9  
Sum of A and B matrices :  
2 4 6  
8 10 12  
14 16 18
```

Q22. What are Dynamic Arrays ?**Ans :**

An array created at compile time by specifying size of the source code has a fixed size and cannot be modified at runtime. 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*.

In C it is possible to allocate memory to arrays at run time. This feature is known as *dynamic memory allocation* and the arrays created at run time are called *dynamic arrays*. This effectively postpones the array definition to run time.

A dynamic array is not the same thing as a dynamically-allocated array, which is a fixed-size array whose size is fixed when the array is allocated.

The simplest dynamic array is constructed by allocating a fixed-size array and then dividing it into two parts :

1. The first stores the elements of the dynamic array and
2. The second is reserved, or unused.

We can then add or remove elements at the end of the dynamic array in constant time by using the reserved space, until this space is completely consumed. The number of elements used by the dynamic array contents is its *logical size* or *size*, while the size of the underlying array is called the dynamic array's *capacity*, which is the maximum possible logical size.

The dynamic array has **performance** similar to an array, with the addition of new operations to add and remove elements from the end:

- Getting or setting the value at a particular index (constant time)
- Iterating over the elements in order (linear time, good cache performance)
- Inserting or deleting an element in the middle of the array (linear time)
- Inserting or deleting an element at the end of the array (constant amortized time)

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory use), and random access. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures.

Dynamic arrays are created using what are known as pointer variables and memory management function malloc, calloc and realloc. These functions are included in the header file <stdlib.h>. The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues.

Dynamic memory allocation allows the program to react, while it is executing to the demands of memory such as user input. All the functions for handling dynamic memory allocation require the header file alloc.h.

The various dynamic memory functions are :

- malloc ()
- calloc ()
- realloc ()
- free ()

2.7 CHARACTER ARRAY

Q23. What are strings ? Explain with examples ?

(OR)

Explain about character Array.

Ans :

(June-19, Dec.-18, Imp.)

A string is a sequence of characters. Any sequence or set of characters defined within double quotation symbols is a constant string.

In C it is required to do some meaningful operations on strings they are :

- Reading string displaying strings
- Combining or concatenating strings
- Copying one string to another.
- Comparing string & checking whether they are equal
- Extraction of a portion of a string

Strings are stored in memory as ASCII codes of characters that make up the string appended with '\0' (ASCII value of null). Normally each character is stored in one byte, successive characters are stored in successive bytes.

Character	m	y		a	g	e		i	s
ASCII Code	77	121	32	97	103	10	32	105	115

Character		2		(	t	w	o	)	\0
ASCII Code	32	50	32	40	116	119	41	0	0

The last character is the null character having ASCII value zero.

2.7.1 Declaring and Initializing Strings

Q24. Explain the process of declaring and initializing strings.

Ans :

(June-19)

C 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 name[25];
```

```
char city[10];
```

The initialization of a string must have the following form which is simpler to one dimension array.

```
char month1[ ] = { 'j','a','n','u','a','r','y' };
```

Then the string month is initializing to January. This is perfectly valid but C offers a special way to initialize strings. The above string can be initialized `char month1[ ] = "January";`

The characters of the string are enclosed within a pair of double quotes. The compiler takes care of string enclosed within a pair of a double quotes. The compiler takes care of storing the ASCII codes of characters of the string in the memory and also stores the null terminator in the end.

```
/* String.c string variable */
#include < stdio.h >
main( )
{
    char month[15];
    printf ("Enter the string");
    gets (month);
    printf ("The string entered is %s", month);
}
```

In this example string is stored in the character variable month the string is displayed in the statement.

```
printf("The string entered is %s", month);
```

It is one-dimension array. Each character occupies a byte. A null character (\0) that has the ASCII value 0 terminates the string. The figure below shows the storage of string January in the memory recall that \0 specifies a single character whose ASCII value is zero.

J
A
N
U
A
R
Y
\0

Character string is terminated by a null character '\0'.

A string variable is any valid C variable name & is always declared as an array. 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.

Example :

```
char month[10];
char address[100];
```

The size of the array should be one byte more than the actual space occupied by the string since the compiler appends a null character at the end of the string.

2.7.2 Reading Strings from the Terminal

Q25. Explain reading and writing of strings with examples ?

Ans :

scanf function is that it terminates its input on the first white space it finds. A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

NEW DELHI

then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the reading of string.

The **scanf** function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character. Note that unlike previous **scanf** calls, in the case of character arrays, the ampersand (&) is not required before the variable name.

The **address** array is created in the memory as shown below :

N	E	W	\0	?	?	?	?	?	?
0	1	2	3	4	5	6	7	8	9

Note that the unused locations are filled with garbage.

If we want to read the entire line "NEW DELHI", then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

with the line of text

NEW DELHI

will assign the string "NEW" to **adr1** and "DELHI" to **adr2**.

/* Program reads four words and displays them on the screen */

```
main( )
{
    char word1[40], word2[40], word3[40], word4[40];
    printf("Enter text : \n");
    scanf("%s %s", word1, word2);
    scanf("%s", word3);
    scanf("%s", word4);
    printf("\n");
    printf("word1 = %s\nword2 = %s\n", word1, word2);
    printf("word3 = %s\nword4 = %s\n", word3, word4);
}
```

Output :

```

Enter text :
Johnville Street, New Delhi 01126
word1 = Johnville
word2 = Street,
word3 = NewDelhi
word4 = 01126

```

Reading a Line of Text

C supports a format specification known as the *edit set conversion code* %[. .] that can be used to read a line containing a variety of characters, including whitespaces. For example, the program segment

```

char line [80] ;
scanf ("%[^\\n]", line) ;
printf ("%s", line) ;

```

will read a line of input from the keyboard and display the same on the screen. We would very rarely use this method, as C supports an intrinsic string function to this job.

Writing Strings to Screen

The printf statement along with format specifier %s to print strings on to the screen. The format %s can be used to display an array of characters that is terminated by the null character for example

```
printf ("%s", name);
```

can be used to display the entire contents of the array name.

We can also specify the precision with which the array is displayed. For instance, the specification

```
%12.6
```

indicates that the *first six* characters are to be printed in a field width of 12 columns.

However, if we include the minus sign in the specification (e.g., %-12.6s, the string will be printed left-justified. The program illustrates the effect of various % specifications.

The program and its output are shown in below. The output illustrates the following features of the %s specifications.

1. When the field width is less than the length of the string, the entire string is printed.
2. The integer value on the right side of the decimal point specifies the number of characters to be printed.
3. When the number of characters to be printed is specified as zero, nothing is printed.
4. The minus sign in the specification causes the string to be printed left-justified.
5. The specification %.ns prints the first n characters of the string.

Program to store the string "United States" in the array country and display the string under various format specifications.

```

main(.)
{
    char country[15] = "United States" ;
    printf ("\n\n") ;
    printf ("*123456789012345*\n");
    printf ("-----\n");
    printf ("%15s\n", country);
    printf ("5s\n", country);
    printf ("15.6s\n", country);
    printf ("% -15.6s\n", country);
    printf ("%15.0s\n", country);
    printf ("% .3s\n", country);
    printf ("""%s\n", country);
    printf ("-----\n");
}

```

Output : *123456789012345*

```

-----
United States
United States
        United
United

```

Uni
United States

The **printf** on UNIX supports another nice feature that allows for variable field width or precision. For instance

```
printf("%*.*s\n", w, d, string);
```

prints the first **d** characters of the string in the field width of **w**.

Program illustrates different variable field specification by printing sequence of characters.

```
main( )
{
    int c, d;
    char string[ ] = "CProgramming";
    printf("\n\n");
    printf("-----\n");
    for (c = 0; c <= 11 ; c++)
    {
        d = c + 1 ;
        printf("{%-12.*s\n", d, string);
    }
    printf("| ----- | \n");
    for (c = 11 ; c >= 0 ; c-- )
    {
        d = c + 1;
        printf(" |%-12.*s\n", d, string) ;
    }
    printf("----- \n");
}
```

Output :

C
CP
CPr
CPro
CProg
CProgr
CProgra

CProgram
CProgramm
CProgrammi
CProgrammin
CProgramming
CProgramming
CProgrammin
CProgrammi
CProgramm
CProgram
CProgra
CProgr
CProg
CPro
CPr
CP
C

2.7.3 Arithmetic Operations on Characters

Q26. Explain the arithmetic operation on strings.

Ans :

We can also manipulate the characters as we manipulate numbers in c language. When ever the system encounters the character data it is automatically converted into a integer value by the system. We can represent a character as a interface by using the following method.

```
X='a';
printf("%d\n",x);
```

Will display 97 on the screen. Arithmetic operations can also be performed on characters for example `x='z'-1;` is a valid statement. The ASCII value of 'z' is 122 the statement the therefore will assign 121 to variable x.

It is also possible to use character constants in relational expressions for example `ch>'a' && ch <= 'z'` will check whether the character stored in variable ch is a lower case letter. A character digit

can also be converted into its equivalent integer value suppose on the expression $a = \text{character} - '1'$; where a is defined as an integer variable & character contains value 8 then $a = \text{ASCII value of } 8 - \text{ASCII value of } '1' = 56 - 49 = 7$.

We can also get the support of the C library function to convert a string of digits into their equivalent integer values the general format of the function is $x = \text{atoi}(\text{string})$ here x is an integer variable & string is a character array containing string of digits.

2.8 FUNCTIONS from ctype.h, string.h

Q27. Write the builtin function in string.h header file.

Ans :

(Dec.-18, Dec.-17, Dec.-16, Imp.)

C language recognizes that string is a different class of array by letting us input and output the array as a unit and are terminated by null character. C library supports a large number of string handling functions that can be used to carry out many of the string manipulations such as :

- Length (number of characters in the string).
- Concatenation (adding two or more strings)
- Comparing two strings.
- Substring (Extract substring from a given string)
- Copy (copies one string over another)

To do all the operations described here it is essential to include string.h library header file in the program.

The C standard library named string.h (<cstring> header in C++) is used to work with C strings. Confusion or programming errors arise when strings are treated as simple data types. Specific functions have to be employed for comparison and assignment such as strcpy for assignment instead of the standard = and strcmp instead of == for comparison.

Operation	Function	Description
Copying	memcpy	Copies a block of memory
	memmove	Move block of memory
	strcpy	Copy string
	strncpy	Copy n number characters from string
Concatenation	strcat	Concatenate strings
	strncat	Append n number of characters from string
Comparison	memcmp	Compare two blocks of memory
	strcmp	Compare two strings
	strcoll	Compare two strings using locale
	strncmp	Compare first n characters of two strings
	strxfrm	Transform string using locale
Searching	memchr	Locate character in block of memory

	strchr	Locate first occurrence of character in string
	strcspn	Get span until character in string
	strpbrk	Locate character in string
	strrchr	Locate last occurrence of character in string
	strspn	Get span of character set in string
	strstr	Locate substring strtok Split string into tokens
Other	memset	Fill block of memory
	strerror	Get pointer to error message string
	strlen	Get string length

Table : Functions included in <cstring>

strlen() function

This function counts and returns the number of characters in a string. The length does not include a null character.

Syntax n = strlen(string);

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

Example : length = strlen("Hollywood");

The function will assign number of characters 9 in the string to a integer variable length.

/*C program to find the length of the string using strlen() function*/

```
#include <stdio.h>
#include <string.h>
void main( )
{
    char name[100];
    int length;
    printf("Enter the string");
    gets(name);
    length = strlen(name);
    printf("\nNumber of characters in the string is=%d", length);
}
```

strcat() function

When you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The strcat() function joins 2 strings together. It takes the following form

strcat(string1, string2)

string1 & string2 are character arrays. When the function strcat is executed string2 is appended to string1. the string at string2 remains unchanged.

Example

```
strcpy(string1, "Narayan");  
strcpy(string2, "Reddy");  
Printf("%s", strcat(string1, string2);
```

From the above program segment the value of string1 becomes NarayanReddy. The string at str2 remains unchanged as Reddy.

strcmp Function

In C you cannot directly compare the value of 2 strings in a condition like `if(string1 == string2)`.

Most libraries however contain the `strcmp( )` function, which returns a zero if 2 strings are equal, or a non zero number if the strings are not the same. The syntax of `strcmp( )` is given below :

```
Strcmp(string1, string2)
```

String1 & string2 may be string variables or string constants. String1, & string2 may be string variables or string constants some computers return a negative if the string1 is alphabetically less than the second and a positive number if the string is greater than the second.

Example

`strcmp("Newyork", "Newyork")` will return zero because 2 strings are equal.

`strcmp("their", "there")` will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.

`strcmp("The", "the")` will return 32 which is the numeric difference between ASCII "T" & ASCII "t".

strncmpi() Function

This function is same as `strcmp( )` which compares 2 strings but not case sensitive.

Example

`strncmpi("THE", "the");` will return 0.

strcpy() Function

C does not allow you to assign the characters to a string directly as in the statement `name = "Robert";`

Instead use the `strcpy( )` function found in most compilers the syntax of the function is illustrated below.

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

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

```
strcpy(Name,"Robert");
```

In the above example Robert is assigned to the string called name.

strlwr() Function

This function converts all characters in a string from uppercase to lowercase.

Syntax

```
strlwr(string);
```

For example

```
strlwr("NARAYAN") converts to Narayan.
```

2.8.1 Array of Strings**Q28. Explain array of strings ?**

Ans :

A two-dimensional array of strings can be declared as follows:

```
<data_type> <string_array_name>[<row_size>][<columns_size>];
```

Consider the following example on declaration of a two-dimensional array of strings.

```
char s[5][30];
```

Initialization

Two-dimensional string arrays can be initialized as shown

```
char s[5][10]={"Cow","Goat","Ram","Dog","Cat"};
```

which is equivalent to

```
s[0] C o w \0
```

```
S[1] G o a t \0
```

```
S[2] R a m \0
```

```
S[3] D o g \0
```

```
S[4] C a t \0
```

Here every row is a string. That is, `s[i]` is a string. Note that the following declarations are invalid.

```
char s[5][ ] = {"Cow","Goat","Ram","Dog","Cat"};
```

```
char s[ ][ ] = {"Cow","Goat","Ram","Dog","Cat"};
```

Manipulating String Arrays

The following program demonstrates how an individual string of an array of strings can be used to take input from the user. As mentioned before, each row (i.e., `s[i]`, if 's' is the array of strings) of an array of strings is a string.

```
#include <stdio.h>

int main()
{
    int i;
    char s[10][30], t[30];
    for(i=0; i<10; i++)
```

```
scanf("%s", s[i]);  
for(i=0; i<10; i++)  
printf("\n%s", s[i]);  
return 0;  
}
```

The following codes show how arrays of strings may be manipulated. This program checks whether a number is odd or even without using any control statement.

```
#include <stdio.h>  
  
int main()  
{  
    char s[2][5] = {"EVEN", "ODD"};  
    int n;  
    printf("\n enter the number:");  
    scanf("%d",&n);  
    printf("\n The number is %s",s[n%2]);  
    return 0;  
}
```

Rahul Publications

Short Question & Answers

1. What is the use of `scanf()`?

Ans :

`scanf()` Function

`scanf()` is used to read information from the standard input, which is generally the keyboard. Recall that a call to **`scanf()`** contains a string argument and may include additional arguments. For each placeholder – that is, each character sequence beginning with % (but not with %) – and additional argument is included in the call. Recall that additional arguments to **`scanf()`** must be pointers (addresses).

When **`scanf()`** is called, the input is processed and is matched against the string argument. As long as the processing produces matches, the processing of the input line continues.

- If a placeholder is encountered in the string argument to `scanf()`, a value of the specified type as the next element in the input constitutes a match.
- If one or more whitespace characters are encountered in the string argument, any sequence of one or more whitespace characters in the input constitute a match.
- If any other character is encountered in the string argument to `scanf()`, the occurrence of this same character in the input constitutes a match.

2. Explain escape sequences in C - languages?

Ans :

Certain nonprinting characters, as well as the backslash (\) and the apostrophe ('), can be expressed in terms of escape sequence. An escape sequence always begins with a backslash and is followed by one or more special characters. An escape sequence is a means of giving special output instructions. For example, a line feed (LF), which is referred to as a new line in C can be represented as \n. Such escape sequence always represents single character, even though they are written in terms of two or more characters.

The commonly used escape sequence are listed below :

Code	Description
\a	bell (beep)
\b	backspace
\t	horizontal tab
\v	vertical tab
\n	new line
\f	form feed
\r	carriage return
\"	double quotation mark
\'	apostrophe
\?	question mark
\\	backslash
\0	null

3. Explain if-else statement.

Ans :

if ... else Control Statement

This control statement works exactly same as the *if* statement that we have discussed above but, here we provide the alternative statements to be executed if the given condition is evaluated to false that is the *else* part.

Syntax

```
if (<condition>)
{
    Statement 1;
    ---
    Statement n;
}
else
{
    Statement 1;
    ---
    Statement n;
}
```

Example

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 100;
    /* check the boolean condition */
    if( a < 20 ) {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    else {
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

Output :

```
a is not less than 20;
value of a is : 100
```

4. Explain Conditional Operator.

Ans :

The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and : and takes three operands. This operator is popularly known as the *conditional operator*. The general form of use the conditional operator is as follows :

Exp1 ? Exp2 : Exp3;

where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

For example, the segment

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

can be written as

```
flag = ( x < 0 ) ? 0 : 1 ;
```

5. Explain do-while loop.

Ans :

do-while loop

The structure of **do-while** loop is similar to while loop. The difference is that in case of **do-while** loop the expression is evaluated after the body the loop is executed. In case of while loop the expression is evaluated before executing body of loop.

The general form of **do-while** statement is

```
do
    statement;
while (expression);
```

where statement is a single statement or compound statement. In contrast to while loop statement (body of loop), do-while loop is executed one or more times.

In do-while loop, first the statement is executed and then the expression is tested. If the expression evaluates to false, the loop terminates and the execution control transfer to the next statement that follows it.

The following program illustrates the use of do-while loop.

```
# include <stdio.h>
main( )
{
    int counter = 1;
    /* loop control variable */
```

```
do
{
    printf("Now the counter value is      %d\n", counter);
    counter++;
}
while(counter <= 10);
printf("End of the loop");
}
```

Here, the body of loop is repeated 10 times.

In case of *do-while* loop, the statement is executed at least once, whereas in case of while loop, the statement may not execute at all if the expression results false for the first time itself.

6. Explain for loop in C.

Ans :

for Statement

The for loop is most common in major programming languages. However, the for loop in C language is very flexible and very powerful. The for loop is used to repeat the execution of a statement for fixed number of times.

The general form of the for loop is

```
for (initialization; test-condition; increment)
{
    Body of the loop
}
```

Execution process of the for statement is as follows :

- Initialization of the control variables is done first, using assignment statements such as ***i*** = 1 and ***count*** = 0. The variables *i* and *count* are known as loop-control variables.
- The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as *i* < 10 that determines when the loop will exit. If the condition is *true*, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
- When the body of the loop is executed, the control is transferred back to the *for* statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as *i* = *i* + 1 and the new value of the control variable is again tested to see whether it satisfies the loop condition.

If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Consider the following segment of a program:

```
for (x =0; x <= 9; x = x + 1)
{
    printf("%d", x);
}
printf("\n");
```


This for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enclosed within parentheses must be separated by semicolons.

The for statement allows for *negative increments*. For example, the loop shown above can be written as follows :

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

This loop is also executed 10 times, but the output would be from 9 to 0 instead of 0 to 9. Note that braces are optional when the body of the loop contains only one statement.

Program to find the sum of 10 given numbers using for statement.

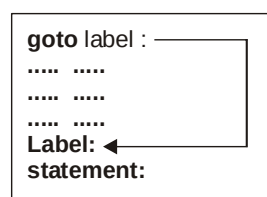
```
#include <stdio.h>
main( )
{
    int number, i, sum = 0;
    printf("Enter values for 10 numbers \n");
    for( i = 1, i < 10; i++ )
    {
        scanf("%d\n", &number);
        sum = sum + number;
    }
    printf("Sum if %d\n", sum);
}
```

7. What is the use of goto statement in C ?

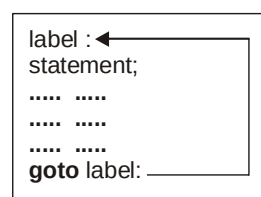
Ans :

C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the *goto* might be desirable.

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



Backward jump

The *label* can be anywhere in the program either before or after the **goto** label; statement. During running of a program when a statement like

```
goto begin ;
```

is met, the flow of control will jump to the statement immediately following the label begin:. This happens unconditionally.

8. Define Array.*Ans :*

C programming language allows working with collection of data elements of same datatype, called as an array. The elements of an array are stored sequentially in memory. In C, string are stored as an array of characters, terminated with a null character.

In C we can define variable called grades, which represents not a single value of grade but a entire set of grades. Each element of the set can then be referenced by means of a number called as index number or subscript.

9. Define Multidimensional Arrays.*Ans :*

Often there is a need to store and manipulate two dimensional data structure such as matrices & tables. Here the array has two subscripts. One subscript denotes the row & the other the column.

The declaration of two dimension arrays is as follows :

```
data_type array_name[row_size][column_size];
int m[10][20]
```

Here *m* is declared as a matrix having 10 rows(numbered from 0 to 9) and 20 columns (numbered 0 through 19). The first element of the matrix is *m*[0][0] and the last row last column is *m*[9][19].

Elements of Multidimensional Arrays

A 2 dimensional array marks [4][3] is shown below figure. The first element is given by marks [0][0] contains 35.5 & second element is marks [0][1] and contains 40.5 and so on.

marks [0][0] 35.5	Marks [0][1] 40.5	Marks [0][2] 45.5
marks [1][0] 50.5	Marks [1][1] 55.5	Marks [1][2] 60.5
marks [2][0]	Marks [2][1]	Marks [2][2]
marks [3][0]	Marks [3][1]	Marks [3][2]

10. What are the uses of Dynamic Array?*Ans :*

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory use), and random access. They usually have only a small fixed additional overhead for storing information about the size and capacity. This makes dynamic arrays an attractive tool for building cache-friendly data structures.

Dynamic arrays are created using what are known as pointer variables and memory management function malloc, calloc and realloc. These functions are included in the header file <stdlib.h>. The concept of dynamic arrays is used in creating and manipulating data structures such as linked lists, stacks and queues.

Dynamic memory allocation allows the program to react, while it is executing to the demands of memory such as use input. All the functions for handling dynamic memory allocation require the header file `alloc.h`.

The various dynamic memory functions are :

- `malloc ( )`
 - `calloc ( )`
 - `realloc ( )`
 - `free ( )`
-

11. Define string and explain string declaration.

Ans :

C 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 name[25];
```

```
char city[10];
```

The initialization of a string must have the following form which is simpler to one dimension array.

```
char month1[ ] = { 'j','a','n','u','a','r','y' };
```

Then the string `month` is initializing to `January`. This is perfectly valid but C offers a special way to initialize strings. The above string can be initialized `char month1[ ] = "January";`

12. How can we combine two strings.

Ans :

strcat() function

When you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The `strcat()` function joins 2 strings together. It takes the following form

```
strcat(string1, string2)
```

`string1` & `string2` are character arrays. When the function `strcat` is executed `string2` is appended to `string1`. the string at `string2` remains unchanged.

Example

```
strcpy(string1, "Narayan");
```

```
strcpy(string2, "Reddy");
```

```
Printf("%s", strcat(string1, string2);
```

From the above program segment the value of `string1` becomes `NarayanReddy`. The string at `str2` remains unchanged as `Reddy`.

13. What is the use of strcpy()?

Ans :

strcpy() Function

C does not allow you to assign the characters to a string directly as in the statement `name = "Robert";`

Instead use the `strcpy( )` function found in most compilers the syntax of the function is illustrated below.

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

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

```
strcpy(Name,"Robert");
```

In the above example Robert is assigned to the string called `name`.

Rahul Publications

Choose the Correct Answers

1. Break statement is used for [b]
(a) Quit a program (b) Quit the current iteration
(c) Both of above (d) None of above
2. Continue statement used for [b]
(a) To continue to the next line of code
(b) To stop the current iteration and begin the next iteration from the beginning
(c) To handle run time error
(d) None of above
3. Due to variable scope in c [a]
(a) Variables created in a function cannot be used in another function
(b) Variables created in a function can be used in another function
(c) Variables created in a function can only be used in the main function
(d) None of above
4. What does the following declaration mean?
int (*ptr)[10]; [b]
(a) ptr is array of pointers to 10 integers
(b) ptr is a pointer to an array of 10 integers
(c) ptr is an array of 10 integers
(d) ptr is a pointer to array
5. The address of the starting element of an array is [c]
(a) Represented by subscripted variable of the starting element
(b) Can't be specified
(c) Represented by the array name
(d) Represented by the size of the array
6. A set of names can be represented as a [a, b, c]
(a) two-dimensional array of characters
(b) one-dimensional array of strings
(c) one-dimensional array of pointers to character
(d) none of these
7. If arr is a two dimensional array of 10 rows and 12 columns, then arr[5] logically points to the [a]
(a) sixth row (b) fifth row
(c) fifth column (d) sixth column

8. Let x be an array. Which of the following operations are illegal ? [a, c, d]
(a) $++x$ (b) $x + 1$
(c) $x++$ (d) $x * 2$
9. In a for loop, if the condition is missing, then, [b]
(a) it is assumed to present and taken to be false
(b) it is assumed to present and taken to be true
(c) it results in a syntax error
(d) execution will be terminated abruptly
10. If switch feature is used, then [c]
(a) default case must be present
(b) default case, if used, should be the last case
(c) default case, if used, can be placed anywhere
(d) none of the above

Rahul Publications

Fill in the Blanks

1. A program loop is a _____ which are repeatedly executed for certain number of times.
2. The while loop is also referred to as _____ .
3. At least _____ times the body of the do-while loop is executed.
4. The for loop structure is a _____ loop.
5. The keyword _____ is used for transferring the control from one statement to other.
6. the if-else statement takes care of _____ as well as _____ conditions.
7. Same condition may be repeated for a number of times in _____ structure.
8. In switch, character constants are automatically converted to _____.
9. Each array element is stored in separate _____ .
10. In an array x[10] the x represents the _____.
11. Array element of two dimensions are stored at _____.
12. Array element counting starts from zero hence the statement x[0] defined can hold _____.
13. Fast access of array elements can be done using _____.
14. A character array always ends with _____.
15. In long k[4] the total memory occupied by the array is _____.

ANSWERS

1. defined block of statements
2. entry controlled loop
3. one
4. pretested
5. goto
6. true, false
7. if ... else
8. integers
9. Memory Locations
10. base address
11. Subsequent memory locations
12. Nothing
13. Pointers
14. null('\0') character
15. 16

UNIT III

Functions: Concept of Function, Using Functions, Call-by-Value Vs Call-by-reference, Passing Arrays to Functions, Scope of Variables, Storage Classes, Inline Functions, and Recursion.

Pointers in C: Introduction, Address of Operator (&), Arrays and Pointers, Pointers and Strings, Pointers to Pointers, Array of Pointers, Pointer to Array, Dynamic Memory Allocation

3.1 FUNCTIONS

3.1.1 Concept of Functions

Q1. What are functions ?

Ans : (Dec.-18)

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

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

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

A function that is built into an application and can be accessed by end-users are called built-in function.

C functions can be classified into two categories :

1. Library functions – printf and scanf are library functions.
2. User-defined functions – main, sqrt, strcat etc. are user-defined functions.

The main difference between these two are that the 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

program. User-defined function can later become a part of the C program library.

Need of Functions

main is a specially recognized function in C. Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These subprograms called 'functions' are much easier to understand, debug, and test.

There are times when certain type of operations or calculations is repeated at many points throughout a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements wherever they are needed.

This "division" approach clearly results in a number of advantages.

1. It facilitates top-down modular programming is shown in fig. below. In this programming style, the high level logic of the overall problem is solved first while the details or each lower-level function are addressed later.
2. The length of a source program can be reduced by using functions at appropriate

places. This factor is particularly critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

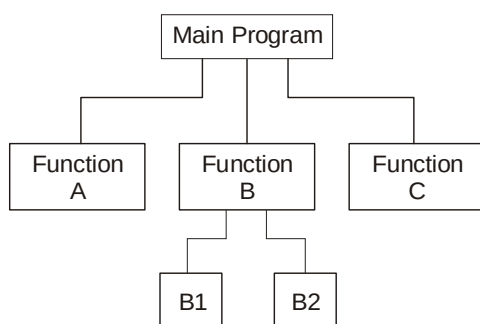


Fig.: Top-down Modular Programming using Functions

3.1.2 Elements of Functions

Q2. What are the advantages and disadvantages of functions ?

Ans :

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

There are many advantages in using functions in a program they are :

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigation.
4. A function may be used by many other programs this means that a C programmer can build on what others have already done, instead of starting over from scratch.
5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.

Functions are used in C for the following reasons :

1. Many programs require that a specific function is repeated many times instead of writing the function code as many times as it is required we can write it as a single function and access the same function again and again as many times as it is required.
2. We can avoid writing redundant program code of some instructions again and again.
3. Programs with using functions are compact & easy to understand.
4. Testing and correcting errors is easy because errors are localized and corrected.
5. We can understand the flow of program, and its code easily since the readability is enhanced while using the functions.
6. A single function written in a program can also be used in other programs also.

Q3. Explain the elements of functions.

Ans :

Elements of Functions

Functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs.

It is therefore not a surprise to note that there exist some similarities between functions and variables in C.

1. Function names and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
2. Same as variables, functions have types associated with them.
3. Same as variables, function names and their types must be declared and defined before they are used in a program.

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition
2. Function call
3. Function declaration

The *function definition* is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the *function call*.

The program (or a function) that calls the function is referred to as the *calling program* or *calling function*. The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the *function declaration* or *function prototype*.

3.1.3 Defining Functions

Q4. Explain the implementation of functions in C - language ?

Ans : (June-17)

A function definition, also known as function implementation shall include the following elements.

1. Function name
2. Function type
3. List of parameters
4. Local variable declarations

5. Function statements

6. A return statement

A general format of a function definition to implement these two parts is given below:

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

The first line

function_type function_name(parameter list)

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

Function Header

The function header consists of three parts: The function type (also known as *return type*, the function name and the *formal* parameter list. Note that a semicolon is not used at the end of the function header.

Name and Type

The *function type* specifies the type of value (like *float* or *double*) 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 it is an integer type. If the function is not returning anything, then we need to specify the return type as *void*. Remember, *void* is one of the fundamental data types in C.

The *function name* in any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be approximate to the task performed by the function.

Formal Parameter List

The *parameter list* declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as *formal* parameters. These parameters can also be used to send values to the calling programs. The parameters are also known as *arguments*.

The parameters list contains declaration of variables separated by commas and surrounded by parentheses.

Examples

```
float quadratic (int a, int b, int c) { . . . }
double power (double x, int n) { . . . }
float mul (float x, float y) { . . . }
int sum (int a, int b) { . . . }
```

There is no semicolon after the closing parenthesis, the declaration of parameter variables cannot be combined. i.e., **int sum (int a, b)** is illegal

A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword void between the parentheses as in

```
void printline (void)
{
    ... ..
}
```

This function neither receives any input values nor returns back any value. Many compilers accept an empty set of parentheses, without specifying anything as in

```
void printline( )
```

Function Body

The *function body* contains the declarations and statements necessary for performing the

required task. The body enclosed in braces, contains three parts, in the order given below:

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

If a function does not return any value (like the `printline` function), we can omit the return statement. However, note that its return type should be specified as `void`. Again, it is nice to have a return statement even for void functions.

3.1.4 Return Values and Their Types

Q5. Write short note on 'Return' keyword in functions ? and types.

Ans :

A function may or may not send back any value to the calling function. If it does, it is done through the return statement. While it is possible to pass to the called function any number of values, the called function can only return *only value* per call, at the most.

The return statement can take one of the following forms :

```
return ;
or
return(expression) ;
```

The 'plain' return does not return any value; it acts much as the closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

An example of the use of a simple return is as follows :

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

The second form of return with an expression returns the value of the expression. For example, the function.

```
int mu1 ( int x, int y )
{
    int p ;
    p = x*y ;
    return (p) ;
}
```

returns the value of p which is the product of the values of x and y. The last two statements can be combined into one statement as follows :

```
return (x*y) ;
```

A function may have more than one return statements. This situation arises when the value returned is based on certain conditions.

For example :

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

All functions by default return *int* type data. But what happens if a function must return some other type? We can force a function to return a particular type of data by using a *type specifier* in the function header as discussed earlier.

When a value is returned, it is automatically cast to the function's type. In functions that do computations using doubles, yet return ints, the returned value will be truncated to an integer.

For instance, the function

```
int product (void)
{
    return (2.5 * 3.0) ;
}
```

will return the value 7, only the integer part of the result.

3.1.5 Function Calls

Q6. Explain how function calls & Declaration is done in C - language ?

Ans :

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

Example

```
main( )
{
    int y ;
```

```
y = mul (10, 5) ;    /* Function call */
printf ("%d\n", y) ;
}
```

When the compiler encounters a function call, the control is transferred to the function `mul()`. This function is then executed line by line as described and a value is returned when a `return` statement is encountered. This value is assigned to `y`.

```
main( )
{
    int y ;
    y = mul (10, 5) ;    /* Function call */
}

int mul(int x, int y)
{
    int p;               /* Local variables */
    p = x * y;           /* x = 10, y = 5 */
    return(p);
}
```

The function call sends two integer values 10 and 5 to the function.

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

which are assigned to `x` and `y` respectively. The function computes the product `x` and `y`, assign the result to the local variable `p` and then returns the value 25 to the main where it is assigned to `y` again. There are many different ways to call a function.

A function which returns a value can be used in expressions like any other variable. Each of the following statements is valid :

```
printf ("%d\n", mul (p, q) ) ;
y = mul (p, q) / (p+q) ;
if (mul (m, n)>total) printf("large");
```

However, a function cannot be used on the right side of an assignment statement.

For instance, **`mul(a, b) = 15;`** is invalid.

A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function.

Function Declaration

All functions in a C program must be declared, before they are invoked. A *function declaration* (also known as *function prototype*) consists of four parts.

1. Function type (return type)

2. Function name
3. Parameter list
4. Terminating semicolon

They are coded in the following format :

```
Function-type function-name (parameter list);
```

This is very similar to the function header line except the terminating semicolon. For example, mul function defined in the previous section will be declared as:

```
int mul (int m, int n); /* Function prototype */
```

Equally acceptable forms of declarations of mul functions are :

```
int mul(int, int);  
mul(int a, int b);  
mul(int, int);
```

When a function does not take any parameters and does not return any value, its prototype is written as :

```
void display (void);
```

A prototype declaration may be placed in two places in a program.

1. Above all the functions (including main)
2. Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a *global prototype*. Such declarations are available for all the functions in the program.

When we place it in a function (in the local declaration section), the prototype is called a *local prototype*. Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the *scope* of the function.

3.1.6 No Arguments and No Return Values

Q7. Explain no arguments and no return values ?

Ans :

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. In effect, there is no data transfer between the calling function and the called function. Fig. below illustrates this. The dotted lines indicate that there is only a transfer of control but not data.

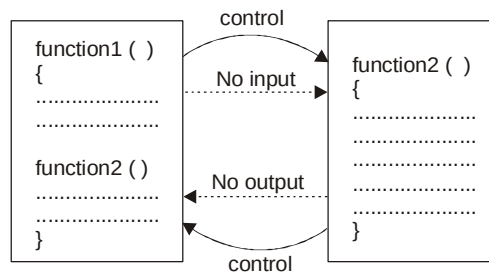


Fig.: No data communication between functions

Example

Program illustrates multiple functions that do not communicate any data between them.

```

/* Function declaration */
void printline (void) ;
void value (void) ;
main( )
{
    printline ( ) ;
    value ( ) ;
    printline ( ) ;
}

/*      Functional: printline ( )      */
void printline (void)                                     /* contains no arguments */
{
    int i ;
    for (i = 1; i <= 35; i++)
        printf("%c", '-' ) ;
    printf("\n") ;
}

/*      Function2: value ( )      */
void value(void)                                         /* contains no arguments */
{
    int year,  period;
    float  inrate,  sum,  principal;
    printf("Principal amount?") ;
    scanf("%f", &principal) ;
    printf("Interest rate?  ") ;
    scanf("%f", &inrate) ;
    printf("Period?      ") ;
    scanf("%d",  &period) ;
    sum = principal ;
}
  
```

```

    year = 1 ;
    while(year <= period)
    {
        sum = sum *(1+inrate) ;
        year = year +1;
    }
    printf("\n%8.2f  %5.2f  %2f  %5d  %12.2f\n", principal , inrate, period, sum) ;
}

```

Output :

```

Principal amount?5000
Interest rate? 0.12
Period? 5
5000.00 0.12 5 8811.71

```

main is the calling function that calls *printline* and *value* functions. Since both the called functions contain no arguments, there are no argument declarations. The *printline* function, when encountered, prints a line with a length of 35 characters as prescribed in the function. The *value* function calculates the value of principal amount after a certain period of years and prints and the results. The following equation is evaluated repeatedly :

$$\text{value} = \text{principal}(1 + \text{interest-rate})$$

The input data include principal amount, interest rate and the period for which the final value is to be calculated. The *while* loop calculates the final value and the results and printed by the library function *printf*. When the closing brace of *value()* is reached, the control is transferred back to the calling function *main*. Return types of both *printline* and *value* are declared as *void*.

3.1.7 Arguments with No Return Values

Q8. Explain about arguments with no return values ?

Ans :

In the above program the main function has no control over the way the functions receive input data. For example, the function *printline* will print the same line each time it is called. Same is the case with the function *value*. We could make the calling function to read data from the terminal and pass it on to the called function.

The nature of data communication between the calling function and the called function with arguments but no return value is shown in figure below.

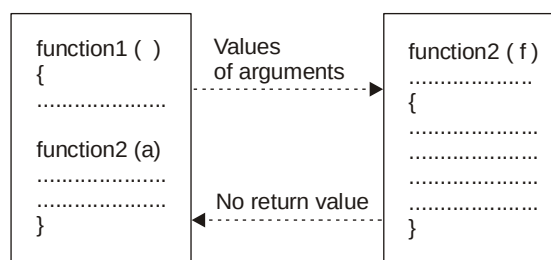


Fig.: One-way Data Communication

We shall modify the definitions of both the called functions to include arguments as follows:

```
void printline(char ch)
```

```
void value(float p, float r, int n)
```

The arguments *ch*, *p*, *r*, and *n* are called the *formal arguments*. The calling function can now send values to these arguments using functions calls containing appropriate arguments. For example, the function call

```
value(500, 0.12, 5)
```

would send the values 500, 0.12 and 5 to the function

```
void value (float p, float r, int n)
```

and assign 500 to *p*, 0.12 to *r* and 5 to *n*. The values 500, 0.12 and 5 are the *actual arguments*, which become the *formal arguments* inside the called function.

The *actual* and *formal* arguments should match in number, type, and order. The values of actual arguments are assigned to the formal arguments on a *one to one* basis, starting with the first argument as shown in Figure below.

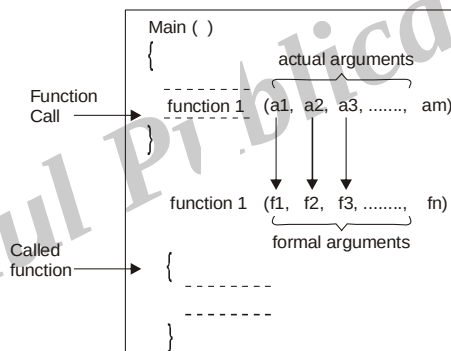


Fig. : Arguments Matching between the Function Call and the Called Function

3.1.8 Arguments with Return Values

Q9. Explain arguments with return values ?

Ans :

To assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such function will have two-way data communication as shown in figure below.

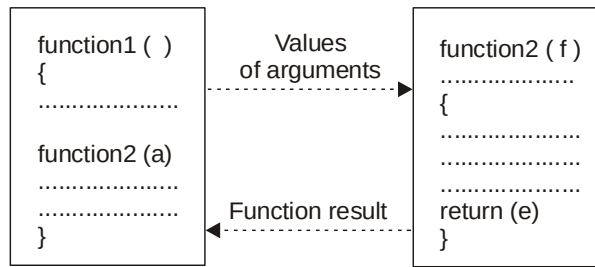


Fig.: Two-way Data Communication between Functions

Program illustrates functions with arguments and return values

```

void printline (char ch, int len) ;
value (float, float, int) ;
main( )
{
    float principal, inrate, amount ;
    int period ;
    printf("Enter principal amount, interest" ) ;
    printf("rate, and period\n") ;
    scanf("%f %f %d", &principal, &inrate, &period) ;
    printline ('*', 52) ;
    amount = value (principal, inrate, period) ;
    printf(" \n % f \ t % f \ t % d \ t % f \ n \ n ", principal,
                                                inrate, period, amount) ;

    printline('=', 52) ;
}
void printline(char ch, int len)
{
    int i ;
    for (i = 1; i <= len; i++) printf ( "%c", ch) ;
    printf ("\n") ;
}
value (float p, float r, int n) /* default return type */
{
    int year ;
    float sum ;
    sum = p ; year = 1 ;
    while(year <=n)
    {
        sum = sum * (1+r) ;
        year = year + 1;
    }
    return (sum) ;          /* returns int part of sum */
}
  
```

Output :

```

Enter principal amount, interest rate, and period
5000    0.12    5
*****
5000.000000    0.1200000    5    8811.000000
*****

```

The calculated value is passed on to main through statement:

```
return(sum);
```

Since, by default, the return type of value function is int, the 'integer' value of sum at this point is returned to main and assigned to the variable amount by the functional call.

```
amount = value (principal, inrate, period);
```

The following events occur, in order, when the above function call is executed.

1. The function call transfers the control along with copies of the values of the actual arguments to the function *value* where the formal arguments *p*, *r* and *n* are assigned the actual value of *principal*, *inrate*, and *period* respectively.
2. The called function *value* is executed line by line in a normal fashion until the *return(sum)*; statement is encountered. At this point, the integer value of *sum* is passed back to the function call in the *main* and the following indirect assignment occurs.

```
value(principal, inrate, period) = sum;
```

3. The calling statement is executed normally and the returned value is thus assigned to *amount*, a *float* variable.
4. Since *amount* is a *float* variable, the returned integer part of sum is converted to floating-point value.

Another important change is the inclusion of second argument to *println* function to receive the value of length of the line from the calling function. Thus, the function call

```
println( '*', 52);
```

will transfer the control to the function *println* and assign the following values to the formal argument *ch*, and *len*;

```
ch = '*';
```

```
len = 52;
```

3.2 USING FUNCTIONS

Q10. Define function ? How can we use Functions in C ?

Ans

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

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

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Parts of functions :

1. Defining a function / Function declaration.
2. Function Definition
3. Function call

1. Defining a Function

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

```
return_type function_name( parameter list ){  
    body of the function  
}
```

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

- **Return Type** ” A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** ” This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** ” A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** ” The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two “

```
/* function returning the max between two numbers */  
int max(int num1,int num2){  
    /* local variable declaration */  
    int result;
```

```
if(num1 > num2)
result= num1;
else
result= num2;
return result;
}
```

2. Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts “

```
return_type function_name( parameter list );
```

For the above defined function max(), the function declaration is as follows “

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration “

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

3. Function call

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

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

Function Arguments

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

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

While calling a function, there are two ways in which arguments can be passed to a function “

3.3 CALL-BY-VALUE VS CALL-BY-REFERENCE

Q11. Explain call by value and call by reference with example ?

Ans :

(Dec.-17, June-17, Imp.)

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

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

While calling a function, there are two ways in which arguments can be passed to a function –

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

Call by Value

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

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

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

Now, let us call the function swap() by passing actual values as in the following example –

```
#include <stdio.h>
/* function declaration */
void swap(int x, int y);
int main() {
    /* local variable definition */
    int a = 100;
    int b = 200;
    printf("Before swap, value of a : %d\n", a);
    printf("Before swap, value of b : %d\n", b);
    /* calling a function to swap the values */
    swap(a, b);
    printf("After swap, value of a : %d\n", a);
}
```

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

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

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

It shows that there are no changes in the values, though they had been changed inside the function.

Call by Reference

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

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

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

Let us now call the function swap() by passing values by reference as in the following example –

```
#include <stdio.h>  
/* function declaration */  
void swap(int *x, int *y);  
int main() {  
    /* local variable definition */  
    int a = 100;  
    int b = 200;  
    printf("Before swap, value of a : %d\n", a);  
    printf("Before swap, value of b : %d\n", b);  
    /* calling a function to swap the values.  
     * &a indicates pointer to a ie. address of variable a and  
     * &b indicates pointer to b ie. address of variable b.  
     */  
}
```

```

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

```

Let us put the above code in a single C file, compile and execute it, to produce the following result –

```

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

```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

3.3.1 Call by Value Vs Call by Reference

Q12. Difference between call by value & call by reference ?

Ans :

(June-19)

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

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

Call by Value	Call by Reference
1. It is the usual method to call a function in which only the value of the variable is passed as an argument.	In this method, the address of the variable is passed as an argument.
2. Any alteration in the value of the argument or parameter passed is local to the function and is not accepted in the calling program.	Any alteration in the value of the argument or parameter passed is accepted in the calling program (since alteration is made indirectly in the memory location using the pointer).
3. A copy of value is passed to the function	An address of value is passed to the function
4. Memory location occupied by formal and actual arguments is different.	Memory location occupied by formal and actual arguments is same and there is a saving of memory location.

5. Since a new location is created, this method is slow.	Since the existing memory location is used through its address, this method is fast.
6. There is no possibility of wrong data manipulation since the arguments are directly used in an expression.	There is a possibility of wrong data manipulation since the addresses are used in an expression. A good skill of programming is needed here.

3.4 PASSING ARRAYS TO FUNCTIONS

Q13. Explain the method of passing arrays to functions ?

Ans :

(Dec.-18, Dec.-17, Imp.)

(i) One-Dimensional Arrays

It is also possible to pass the values of an array to a function. To pass a one-dimensional array to a called function, it is sufficient to list the name of the array, *without any subscripts*, and the size of the array as arguments. For example, the call

largest(a, n)

will pass the whole array a to the called function. The called function expecting this call must be appropriately defined. The largest function header might look like.

float largest(float array[], int size)

The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows :

float array[];

The pair of brackets informs the compiler that the arguments array is an array of numbers. It is not necessary to specify the size of the array here.

Consider a problem of finding the largest value in an array of 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) ;
}

```

When the function call `largest (value, 4)` is made, the values of all elements of array value become the corresponding elements of array `a` in the called function. The `largest` function finds the largest value in the array and returns the result to the main.

By passing the array name, we are, in fact, passing the address of the array to the called function. The array in the called function now refers to the same array stored in the memory. Therefore, any changes in the array in the called function will be reflected in the original array.

Passing address of parameters to the functions is referred to as *pass by address*.

(ii) Two-Dimensional Arrays

We can also pass multidimensional arrays to functions. The approach is similar to the one we did with one-dimensional arrays. The rules are simple.

1. The function must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two dimensions by including two sets of brackets.
3. The size of the second dimension must be specified.
4. The prototype declaration should be similar to the function header.

The function given below calculates the average of the values in a two-dimensional matrix.

```

double average(int x[M] [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) ;
}

```

3.5 SCOPE OF VARIABLES

Q14. Explain the scope of variables ?

Ans :

(June-18, June-17, Imp.)

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

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Local Variables

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

```
#include <stdio.h>
int main() {
    /* local variable declaration */
    int a, b;
    int c;
    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Global Variables

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

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program shows how global variables are used in a program.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main () {
    /* local variable declaration */
    int a, b;
    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
}
```

```
    return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example-

```
#include <stdio.h>
/* global variable declaration */
int g = 20;
int main () {
    /* local variable declaration */
    int g = 10;
    printf ("value of g = %d\n", g);
    return 0;
}
```

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

value of g = 10

Formal Parameters

Formal parameters, are treated as local variables within a function and they take precedence over global variables. Following is an example –

```
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main() {
    /* local variable declaration in main function */
    int a = 10;
    int b = 20;
    int c = 0;
    printf ("value of a in main() = %d\n", a);
    c = sum( a, b);
    printf ("value of c in main() = %d\n", c);
    return 0;
}

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

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

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows :

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Q15. What is the life time of variables ?

Ans :

The *lifetime* of a variable is the interval of time for which the variable exists; i.e. the time from when it is created to when it is destroyed; *duration*, *span*, or *extent* are equivalent terms for the same thing.

It is common to find confusion between *scope* and *lifetime* - though they are in cases related, they are entirely different notions: *lifetime* is to do with a period of time during the execution of a program, *scope* is to do with which parts of a program text. In C and C++ , lifetime is *dynamic* - you must execute the program in order to determine it. Scope is *static* - determinable at compile time, or by reading the program text.

In the case of local variables (local to blocks or functions), and where there are no *scope-holes*, lifetime and scope correspond: scope is the remainder of the block / function after the variable definition; lifetime is the whole time that control is in that part of the program from the definition to the end of the block / function.

Example: Local Variables

```
int fred(int a, int b)
{
    int c; /* when control passes into 'fred': local (AUTOMATIC) variables
           a,b, c are created at this time — their LIFETIME starts then */
    c = 22;
    ...
}
```

/*when control reaches here, locals are destroyed*/

Thus, c, and a, b exist only for the duration of the call to function fred. For each call, entirely new variables are created and destroyed. If you wanted to retain the value of c from call to call, you would have to use the *static* type modifier.

3.6 STORAGE CLASSES

Q16. Explain the use of different storage class specifiers.

(or)

Explain different types of variables available in C ?

Ans :

(Nov.-20, Dec.-18, Dec.-16, Imp.)

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.

We define the storage class of an object using one of four specifiers : auto, register, static, and extern.

There are two different ways to characterize variables: by *data type*, and by *storage class*. Data type refers to the type of information represented by a variable, e.g., integer number, floating-point number, character, etc. Storage class refers to the permanence of a variable, and its scope within the program, i.e., the portion, i.e., the portion of the program over which the variable is recognized.

There are four different storage-class specification in C are :

- 1) *automatic*,
- 2) *external*,
- 3) *static* and
- 4) *register*

They are identified by the keywords auto, extern, static, and register, respectively.

Object Storage Attributes

Storage class specifiers control three attributes of an object's storage as shown in figure below; its scope, extent, and linkage.

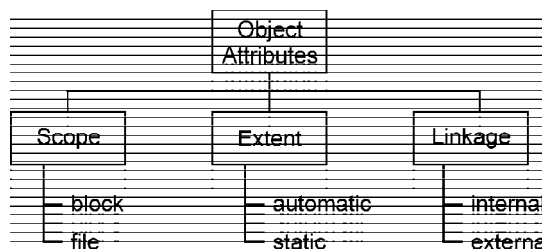


Fig. : Object Storage Attributes

1. Scope

Scope defines the visibility of an object; it defines *where* an object can be referenced. In C, an object can have four levels of scope; block, file, function, and function-prototype. Here, we do not discuss the function and function prototype scopes.

- **Block (Local) Scope :** The body of a function is a block and a compound statement in a loop in a nested block within its function block. When the scope of an object is block, it is visible only if the block in which it is defined. An object with a block scope is sometimes referred to as a local object.

The scope of an object declared anywhere in a block has block scope. For example, a variable declared in the formal parameter list of a function has block scope. A variable declared in the initialization section of a *for* loop also block scope, but only within the *for* statement.

- **File (Global) Scope :** File scope includes the entire source file for a program, including any files included in it. An object with file scope has visibility through the whole source file in which it is declared. However, objects within block scope are excluded from file scope unless specifically declared to have file scope; in other words, by default block scope hides objects from file scope. File scope generally includes all declarations outside a function and all function headers. An object with file scope is sometimes referred to as a global object.

2. Extent

The extent of an object defines the duration for which the computer allocates memory for it. In C, an object can have automatic, static extent, or dynamic extent. The extent of an object is also known as its storage duration.

- **Automatic Extent :** An object with an automatic extent is created each time its declaration is encountered and is destroyed each time its block is exited.

For example, a variable declared in the body of a loop is created and destroyed in each iteration. Note however, that declarations in a function are not destroyed until the function is complete; when the function calls a function, they are out of scope but not destroyed.

- **Static Extent :** A variable with a static extent is created when the program is loaded for execution and is destroyed when the execution stops. This is true no matter how many times the declaration is encountered during the execution.
- **Dynamic Extent :** Dynamic extent is created by the program through the *malloc* and its related library functions. We do not discuss static memory in this text.

3. Linkage

A large application program may be broken into several modules, with each module potentially written by a different programmer. Each module is a separate source file with its own objects. Different modules may be related when the program is linked edited.

We can define two types of linkage: internal and external. As we discuss later, the linkage characteristic differs from scope characteristic, although they look the same at the first glance.

- **Internal Linkage :** An object with an internal linkage is declared and visible only in one module. Other module cannot refer to this object.
- **External Linkage :** An object with an external linkage is declared in one module but is visible in all other modules that declare it with a special keyword, *extern*, as we discuss shortly.

3.6.1 Storage Class Specifiers

Q17. Explain the functions associated with storage classes.

OR

Define storage classes, list & explain various storage classes in C ?

Ans :

(June-18)

We have defined three storage classes – scope, extent, and linkage – with potentially eight different combinations ($2 \times 2 \times 2$). If we want to sue a specifier for each combination, we need a total or eight different specifiers. However, there are only four storage class specifiers, as shown in figure below. auto, register, static, and extern. Three factors have contributed to this reduced set.

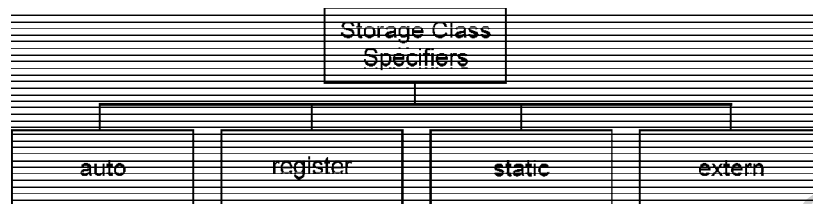


Fig.: Storage Class Specifiers

1. The scope is not defined, it is understood. The place where an object is declared defines the scope.
2. Not all combinations are logically possible.
3. One specifier, static, is used for two different purposes. The scope shows what it defines.

1. **auto - Storage class**

Automatic variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as an automatic variable unless a different storage-class specification is shown within the declaration.

Since the location of the variable declarations within the program determines the automatic storage class, the keyword `auto` is not required at the beginning of each variable declaration. There is no harm in including an `auto` specification within a declaration, though this is normally not done.

An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

auto is the default storage class for all local variables.

```

{
    int Count;
    auto int Month;
}
  
```

The example above defines two variables with the same storage class. `auto` can only be used within functions, i.e. local variables.

2. register - Storage class

Registers are special storage areas within the computer's central processing unit. The actual arithmetic and logical operations that comprise a program are carried out within these registers.

In C, the values of *register variables* are stored within the registers of the central processing unit. A variable can be assigned this storage class simply by preceding the type declaration with the keyword `register`. There can, however, be only a few register variables (typically, two or three) within any one function. The exact number depends upon the particular computer, and the specific C compiler. Usually, only integer variables are assigned the register storage class.

The register and automatic storage classes are closely related. In particular, their visibility (i.e., their scope) is the same. Thus, register variables, like automatic variables, are local to the function in which they are declared. Furthermore, the rules governing the use of register variables are the same as those for automatic variables, except that the address operator (&) cannot be applied to register variables.

```
{  
    register int  Miles;  
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

3. static - Storage class

Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the static storage-class designation. Static variables can be utilized within the function in the same manner as other variables. They cannot, however, be accessed outside of their defining function.

static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count;  
int Road;  
{  
    printf("%d\n", Road);  
}
```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initialized at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls.

```
void func(void);  
static count=10;  
  
/* Global variable - static is the default */
```

```
main( )
{
    while (count-->0)
    {
        func( );
    }
}

void func( void )
{
    static i = 5;
    i++;
    printf("i is %d and count is %d\n", i, count);
}
```

This will produce following result

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

Here keyword *void* means function does not return anything and it does not take any parameter. You can memorise void as nothing. static variables are initialized to 0 automatically.

The rules relating to **static** include :

- Variables declared outside all blocks without the **static** keyword always retain their values throughout the program. To restrict their access to a particular translation unit, you must use the **static** keyword. This gives them "internal linkage." To make them global to an entire program, omit the explicit storage class or use the keyword **extern**. This gives them "external linkage." Internal and external linkage are also discussed in Linkage.
- You can define a variable at the external level only once within a program. You can define another variable with the same name and the **static** storage-class specifier in a different translation unit. Since each **static** definition is visible only within its own translation unit, no conflict occurs. This provides a useful way to hide identifier names that must be shared among functions of a single translation unit, but not visible to other translation units.
- The **static** storage-class specifier can apply to functions as well. If you declare a function **static**, its name is invisible outside of the file in which it is declared.

4. **extern - Storage class**

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another files.

File 1: main.c

```
int count = 5;

main( )
{
    write_extern( );
}
```

File 2: write.c

```
void write_extern(void);
extern int count;
void write_extern(void)
{
    printf("count is %i\n", count);
}
```

Here *extern* keyword is being used to declare count in another file.

Now compile these two files as follows

```
gcc main.c write.c -o write
```

This file produce *write* program which can be executed to produce result.

Count in 'main.c' will have a value of 5. If main.c changes the value of count - write.c will see the new value.

The rules for using **extern** are :

- The **extern** storage-class specifier declares a reference to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible prior to its definition in the same source file. Once you have declared a reference to the variable at the external level, the variable is visible throughout the remainder of the translation unit in which the declared reference occurs.
- For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the external level. This definition (without the **extern** storage class) can be in any of the translation units that make up the program.

3.7 INLINE FUNCTION

Q18. Explain about inline functions ?

Ans : (Dec.-19, June-19, Dec.-17, Imp.)

C99 adds the keyword inline, which applies to functions. By preceding a function declaration with inline, you are telling the compiler to optimize calls to the function. Typically, this means that the function's code will be expanded in line, rather than called. However, inline is only a request to the compiler, and can be ignored. Specifically, C99 states that using inline 'suggests that calls to the function be as fast as possible.' The inline specifier is also supported by C++, and the C99 syntax for inline is compatible with C++.

To create an inline function, precede its definition with the inline keyword. For example, in the following program, calls to the function max() are optimized :

```
#include <stdio.h>
inline int max(int a, int b)
{
    return a > b ? a : b;
}

int main(void)
```

```
{
    int x = 5, y = 10;
    printf("Max of %d and %d is : %d\n", x, y, max(x, y));
    return 0;
}
```

For a typical implementation of inline, the preceding program is equivalent to this one :

```
#include <stdio.h>
int main(void)
{
    int x = 5, y = 10;
    printf("Max of %d and %d is : %d\n", x, y, (x>y ? x : y) );
    return 0;
}
```

The reason that inline functions are important is that they help you create more efficient code while maintaining a structured, function-based approach. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time.

However, when a function is expanded in line, none of these operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to inline only very small functions. Further, it is also a good idea to inline only those functions that will have significant impact on the performance of your program.

3.8 RECURSION FUNCTIONS

Q19. Write a short notes on Recursion Functions.

Ans :

(Nov.-20, June-19, Imp.)

A recursive function is one which calls itself. In C, it is possible for the functions to call themselves. A function is called *recursive* if a statement within the body of a function calls the same function itself.

A very simple example of recursion is presented below :

```
fun( )
{
    printf("This is an example of recursion\n")
    fun( ) ;
}
```

When executed, this program will produce an output something like this :

```
This is an example of recursion
This is an example of recursion
```

This is an example of recursion

This is an example of recursion

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below :

$$\text{factorial of } n = n(n-1)(n-2)\dots\dots\dots 1.$$

For example,

$$\text{factorial of } 4 = 4 \times 3 \times 2 \times 1 = 24$$

A function to evaluate factorial of n is as follows :

```
factorial (int n)
{
    int fact ;
    if (n==1)
        return(1) ;
    else
        fact = n * factorial(n-1) ;
    return (fact) ;
}
```

Let us see how the recursion works. Assume $n = 3$. Since the value of n is not 1, the statement

$$\text{fact} = n * \text{factorial}(n-1) ;$$

will be executed with $n = 3$. That is, $\text{fact} = 3 * \text{factorial}(2)$; will be evaluated. The expression on the right-hand side includes a call to factorial with $n = 2$. This call will return the following value :

$$2 * \text{factorial}(1)$$

Once again, factorial is called with $n = 1$. This time, the function returns 1. The sequence of operations can be summarized as follows :

$$\begin{aligned} \text{fact} &= 3 * \text{factorial}(2) \\ &= 3 * 2 * \text{factorial}(1) \\ &= 3 * 2 * 1 = 6 \end{aligned}$$

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solutions to subsets of the problems. When we write recursive functions. We must have an if statement somewhere to force the function to return without the recursive call being executed.

Program : Recursive Fibonacci Function

```
#include <stdio.h>
long Fibonacci( long n );
```

```
int main( )
{
    long result;
    long number;
    printf( "Enter an integer: " );
    scanf( "%ld", &number );
    result = fibonacci( number );
    printf( "Fibonacci( %ld ) = %ld\n", number, result );
    return 0;
}

long fibonacci( long n )
{
    if ( n == 0 || n == 1 ) {
        return n;
    }
    else {
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    }
}
```

Output :

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

Q20. Explain method of implementation functions with strings.

Ans :

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

Basic rules are :

1. The string to be passed must be declared as a formal argument of the function when it is defined.

Example :

```
void display(char item_name[ ])
{
    ... ..
    ... ..
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument.

Example

```
display(names);
```

where names is a properly declared string array in the calling function. Like arrays, strings in C cannot be passed by value to functions.

Q21. Explain about nested functions with examples ?

Ans :

C permits nesting of two functions freely. There is no limit how deeply functions can be nested. Suppose a function a can call function b and function b can call function c and so on.

Consider the following program:

```
main( )
{
    int a, b, c;
    float ratio( );
    scanf("%d%d%d", &a, &b, &c);
    printf("%f\n", ratio(a, b, c));
}

float ratio(x, y, z) {
    int x, y, z;
    if(difference(y, z))
        return(x/y - z);
    else
        return(0, 0);
}

difference(p, q)
{
    int p, q;
    if(p!=q)
        return(1);
    else
        return(0);
}
```

The above program calculates the ratio $a/b - c$; and prints the result. We have the following three functions :

```
main()
ratio()
difference()
```

main reads the value of a, b, c and calls the function ratio to calculate the value $a/b - c$ this ratio cannot be evaluated if $(b - c)$ is zero. Therefore ratio calls another function difference to test whether the difference $(b - c)$ is zero or not.

3.9 POINTERS INTRODUCTION**Q22. What are pointers ? Explain in detail with examples ?**

Ans : (June-19, Dec.-18, June-17, Imp.)

In C a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They are :

1. Pointers are more efficient in handling arrays and 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 passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such

as structures, linked lists, queues, stacks and trees.

7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

Fundamentals of Pointers

Every stored data item occupies one or more contiguous memory cell (i.e., adjacent words or bytes). The number of memory cell required to store a data item depends on the type of data item. For example, a single character will typically be stored in one byte (8 bits) of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes; and a double-precision quantity may require eight contiguous bytes.

Suppose v is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can then be accessed if we know the location (i.e., the *address*) of the first memory cell.* The address of v 's memory location can be determined by the expression $\&v$, where $\&$ is a unary operator, called the *address operator*, that evaluates the address of its operand.

Now let assign the address of v to another variable, pv . Thus,

$$pv = \&v$$

This new variable is called a *pointer* to v , since it “points” to the location where v is stored in memory. However, that pv represents v 's *address*, not its value. Thus, pv is referred to as a *pointer variable*. The relationship between pv and v is illustrated in Fig. below.

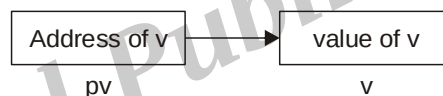


Fig.: Relationship between pv and v
(where $pv = \&v$ and $v = *pv$)

The data item represented by (i.e., the data item stored in v 's memory cells) can be accessed by the expression $*pv$, where $*$ is a unary operator, called the *indirection operator*, that operates only on a pointer variable. Therefore, $*pv$ and v both represent the same data item (i.e., the contents of the same memory cells).

If we write $pv = \&v$ and $u = *pv$, then u and v will both represent the same value; i.e., the value of v will indirectly be assigned to u .

Example

Program illustrates the relationship between two integer variables, their corresponding addresses and their associated pointers.

```
# include <stdio.h>
main( )
{
    int u = 3 ;
    int v ;
    int *pu ; /* pointer to an integer */
    int *pv ; /* pointer to an integer */
}
```



```

    pu = &u ; /* assign address of u to pu */
    v = *pu ; /* assign value of u to v */
    pv = &v ; /* assign address of v to pv */
    printf( "\ nu=%d    &u=%X    pu=%X    *pu=%d", u, &u, pu, *pu) ;
    printf( "\ nnv=%d    &v=%X    pv=%X    *pv=%d", v, &v, pv, *pv) ;
}

```

Note that *pu* is a pointer to *u*, and *pv* is a pointer to *v*. Therefore *pu* represents the address of *u*, and *pv* represents the address of *v*.

Output :

```

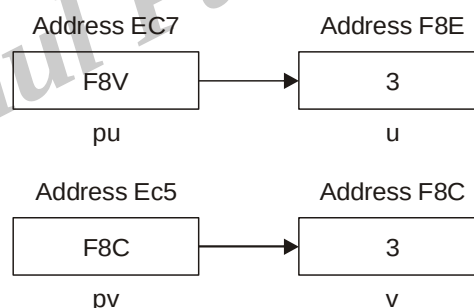
u=3    &u = F8E    pu = F8E    *pu=3
v=3    &v = F8C    pv = F8C    *pv=3

```

In the first line, we see that *u* represents the value 3, as specified in the declaration statement. The address of *u* is determined automatically by the compiler as F8E (hexadecimal). The pointer *pu* is assigned this value; hence, *pu* also represents the (hexadecimal) address F8E. Finally, the value to which *pu* points (i.e., the value stored in the memory cell whose address is F8E) is 3, as expected.

The second line shows that *v* also represents the value 3. This is expected, since we have assigned the value **pu* to *v*. The address of *v*, and hence the value of *pv*, is F8C. *u* and *v* have different addresses. Finally, we see that the value to which *pv* points is 3, as expected.

The relationships between *pu* and *u*, and *pv* and *v*, are shown in fig. below. The memory locations of the pointer variables (i.e., address EC7 for *pu*, and EC5 for *pv*) are not displayed by the program.



Address of Operator (&)

The & is a unary operator in C which returns the memory address of the passed operand. This is also known as address of operator.

- Pointer address operator is denoted by '&' symbol
- When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

&n – It gives an address on variable n

Working of Address Operator

```

#include<stdio.h>
void main()

```

```

{
    int n = 10;
    printf("\nValue of n is : %d",n);
    printf("\nValue of &n is : %u",&n);
}

```

Output :

Value of n is : 10

Value of &n is : 1002

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with **%u**.

```
printf("\nValue of &n is : %u",&n);
```

Understanding Address Operator

Consider the following program –

```

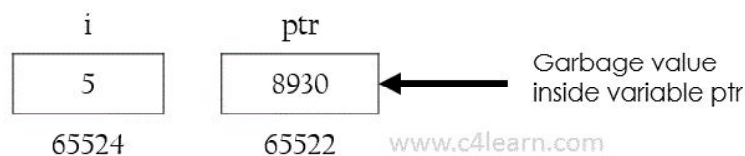
#include<stdio.h>
int main()
{
    int i = 5;
    int *ptr;
    ptr = &i;
    printf("\nAddress of i : %u",&i);
    printf("\nValue of ptr is : %u",ptr);
    return(0);
}

```

After declaration memory map will be like this –

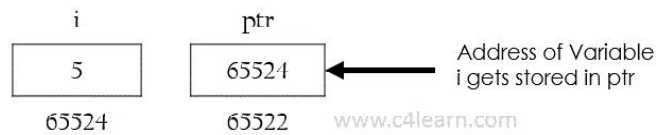
int i = 5;

int *ptr;



after Assigning the address of variable to pointer , i.e after the execution of this statement

```
ptr = &i;
```



3.9.1 Declaring Pointer Variables

Q23. Explain the pointer variables are declared ?

Ans :

(Dec.-16)

Every variable must be declared for its type in C. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

The compiler tells 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 example,

```
int *p;      /* integer pointer */
```

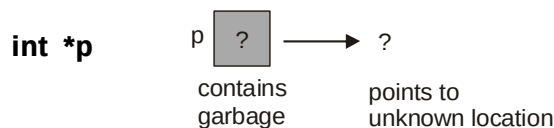
declares the variable `p` as a pointer variable that points to an integer data type. The type ***int*** refers to the data type of the variable being pointed to by `p` and not the type of the value of the pointer. The statement

```
float *x;    /* float pointer */
```

declares `x` as a pointer to a floating-point variable.

The declarations cause the compiler to allocate memory locations for the pointer variables `p` and `x`.

Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown :



3.9.2 Initialization of Pointer Variables

Q24. How pointers initialized ?

Ans :

(Dec.-16)

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

All uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

Example

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

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

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

is allowed. The only requirement here is that the variable **quantity** must be declared before the initialization takes place. Remember, this is an initialization of **p** and ***p**. Ensure that the pointer variables always point to the corresponding type of data. For example.

```
float a, b ;
int x, *p ;
p = &a ;           /* wrong */
b = *p ;
```

will result in erroneous output because we are trying to assign the address of a **float** variable to an integer pointer. When we declare a pointer to be of **int** type, the system assumes that any address that the pointer will hold will point to an integer variable.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

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

is perfectly valid. It declares **x** as an integer variable and **p** as a pointer variable and then initializes **p** to the address of **x**. And also remember that the target variable **x** is declared first.

The statement

```
int *p = &x, x;  is not valid.
```

Q25. Define NULL Pointers ?

Ans :

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

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

```
#include <stdio.h>

int main() {
    int *ptr = NULL;
```

```
printf("The value of ptr is : %x\n", ptr );  
return 0;  
}
```

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

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr) /* succeeds if p is not null */  
if(!ptr) /* succeeds if p is null */
```

Q26. What is Indirection Operator in pointers ?

Ans :

After a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This is done by using another unary operator* (asterisk), usually known as the *indirection 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** and 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 (on the right-hand side of the equal sign), 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**. The * can be remembered as 'value at address'. Thus the value of **n** would be 179. The two statements

```
p = &quantity ;  
n = *p ;
```

are equivalent to

```
n = *&quantity ;
```

which in turn is equivalent to

```
n = quantity ;
```

Program to illustrate the use of indirection operator '*' to access the value pointed to by a pointer.

```
main( )
{
    int  x, y ;
    int  *ptr ;
    x = 10;
    ptr = &x ;
    y = *ptr ;

    printf ( " Value of  x  is %d\n\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("\nNow  x  = %d\n", x) ;
}
```

Output :

```
Value  of  x  is 10
10      is stored at addr  4104
10      is stored at addr  4104
10      is stored at addr  4104
4104    is stored at addr  4106
10      is stored at addr  4108
Now  x  = 25
```

The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

The statement **ptr = &x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y**.

***ptr = 25;**

This statement puts the value of 25 at the memory location whose address is the value of **ptr**. We know that the value of **ptr** is the address of **x** and therefore the old value of **x** is replaced by 25. This, in effect, is equivalent to assigning 25 to **x**. This shows how we can change the value of a variable *indirectly* using a pointer and the *indirection operator*.

3.10 ADDRESS OF OPERATOR

Q27. Explain the types of operators in pointers to indicate address?

Ans :

There are two important operators which are highly required, if you are working with the pointers. Without these operators, we cannot work with the pointers.

The operators are:

1. The * Operator (Dereference Operator or Indirection Operator)
2. The & Operator (Address Of Operator)

1. The * Operator (Dereference Operator or Indirection Operator)

"Dereference Operator" or "Indirection Operator" denoted by asterisk character (*), * is a unary operator which performs two operations with the pointer (which is used for two purposes with the pointers).

- A. To declare a pointer
- B. To access the stored value of the memory (location) pointed by the pointer

A) To declare a pointer

Consider the syntax

```
data_type *pointer_variable_name;
```

Let suppose, if we declare a pointer ptrX to store the memory address of an integer variable; then the pointer declaration will be `int *ptrX;`

B) To access the stored value of the memory (location) pointed by the pointer

Consider the syntax

```
*pointer_variable_name;
```

Let suppose, if there is a pointer variable ptrX which is pointing to the address of an integer variable x; then to access the value of x, `*ptrX` will be used.

2) The & Operator (Address Of Operator)

The "Address Of" Operator denoted by the ampersand character (&), & is a unary operator, which returns the address of a variable.

After declaration of a pointer variable, we need to initialize the pointer with the valid memory address; to get the memory address of a variable Address Of" (&) Operator is used.

Consider the program

```
#include <stdio.h>
int main()
{
    int x=10; //integer variable
    int *ptrX; //integer pointer declaration
```

```
ptrX = &x; //pointer initialization with the address of x
printf("Value of x: %d\n", *ptrX);
return 0;
}
```

Output :

Value of x: 10

3.11 USES OF POINTERS

Q28. List out uses of pointer in C.

Ans :

(Dec.-18, Dec.-17, June-17, Imp.)

Uses

1. Pointers provide direct access to memory
2. It provides a way to return more than one value to the function.
3. Reduces the storage space and complexity of the program.
4. Reduces the execution time of the program.
5. Provides an alternative way to access array elements.
6. It allows us to perform the dynamic memory allocation. And allows us to re-size it too.
7. Address of objects can be extracted using pointers.
8. Pointers help us to build complex data structures like linked lists, queues, stacks, trees, graphs etc.

3.12 POINTERS AND ARRAYS

Q29. Explain array of pointers with example ? (or) Explain about passing array to functions.

Ans :

(June-19, June-18, Imp.)

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 contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

Since a block of 10 integers located contiguously in memory, it is nothing but an array of 10 integers.

The following program illustrate the relationship between pointers and arrays :

```
# include <stdio.h>
main( )
{
    int j ;
    static int a[ ] = {5, 8, 2, 98, 16, 20, -73, 26, 57, 1};
    int *ip;
```



```

ip = &a[0];          /* point ip pointer to the first element of the array */
for ( j = 0; j < 10; j++)
{
    printf( "a[%d] = %d", j, a[ j ] ) ;
    printf( "ip + %d = %d \n", j, * (ip + j)) ;
}
}

```

Here we have an array containing 10 integers. Usually the array elements are referred to means of subscripts i.e., $a[0]$ through $a[9]$. Alternatively, we can access them using pointer as shown in the program. Also the elements of array can be printed either using the carry subscript notation or de-referencing the pointer.

The two printf statements in the above program output the same value. In second printf statement, the j is added to ip , then the new pointer is de-referenced. The reader can test the program by mentioning $*ip++$ and $*(++ip)$ in place of $*(ip + j)$.

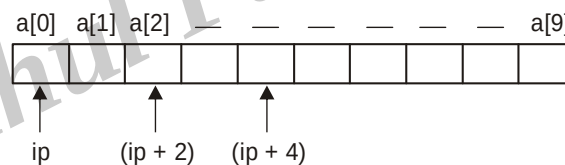
Consider the following statements :

```

int * jp;
int a[10];

```

Standard C allows us to use variable_name in place of $\&\text{variable\_name}[0]$.



Thus, the statements

```
ip = &a[0];
```

and

```
ip = a;
```

result in same. Here, the pointer ip points to first location of array a . That is, now, ip is same as $\&a[0]$.

But, there is an important difference between the same of an array and a pointer variable. We know that, the name of an array is address of first element in the array but neither it can be incremented nor assigned with new value. So,

```
ip++;
```

is legal whereas as

```
a++;
```

is illegal. Similarly, the statement $*ip = t$, where t is a variable, is legal statement whereas $*a = t$ is illegal.

3.13 POINTERS AND CHARACTER STRINGS

Q30. Explain pointers with character strings ? (or) Explain about pointers and strings.

Ans :

(June-18)

Strings are treated like character arrays and therefore they are declared and initialized as follows :

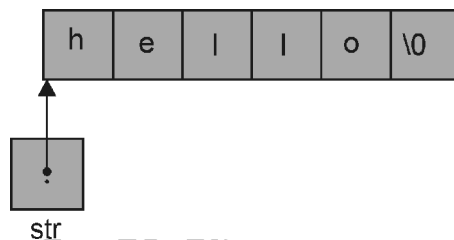
```
char str [5] = hello;
```

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type **char**.

Example

```
char *str = "hello";
```

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 "hello" as :



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

Example

```
Char *string1;
```

```
string1 = "hello";
```

Note that the assignment

```
string = "hello"
```

is not a string copy, because the variable **string1** is a pointer, not a sting.

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

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

```
puts (string1);
```

although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator * here.

Example

Program using pointers to determine the length of a character string.

A program to count the length of a string is shown below. The statement

```
char *cptr = name;
```

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

```
while(*cptr != '\0')
```

 is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr - name;
```

gives the length of the string **name**.

Program to determine the length of a character string using pointers

```
main( )
{
    char *name;
    int length;
    char *cptr = name;
    name = "HELLO";
    printf ("%s\n", name) ;
    while(*cptr != '\0')
    {
        printf( "%c is stored at address %u\n", *cptr, cptr) ;
        cptr++;
    }
    length = cptr - name;
    printf( "\n Length of the string = %d\n", length) ;
}
```

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid :

```
char *name ;
```

```
name = "Hello"
```

These statements will declare **name** as a pointer to character and assigned to **name** the constant character string "Hello". You might remember that this type of assignment does not apply to character arrays.

The statements like

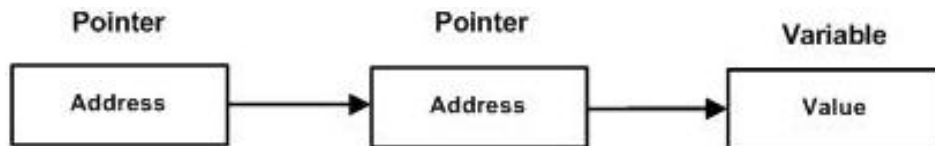
```
char name[20];
```

```
name = "Hello"
```

do not work.

Q31. Explain the initialization of pointer to pointers ?**Ans :****(June-18)**

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int -

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

```
#include <stdio.h>

int main() {
    int var;
    int *ptr;
    int **pptr;
    var = 3000;
    /* take the address of var */
    ptr = &var;
    /* take the address of ptr using address of operator & */
    pptr = &ptr;
    /* take the value using pptr */
    printf("Value of var = %d\n", var);
    printf("Value available at *ptr = %d\n", *ptr);
    printf("Value available at **pptr = %d\n", **pptr);
    return 0;
}
```

Output :

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

3.14 DYNAMIC MEMORY ALLOCATION

Q32. What is DMA ? Explain the related functions with examples ?

Ans :

(Nov.-20, (Dec.-17, June-17, Imp.)

The process of allocating memory during program execution is called **dynamic memory allocation**. In dynamic memory management memory will be allocated at runtime. This is also known as heap memory. Some language at the run time have ability to calculate and assign the memory space at run time for an array but C can't have this feature. But in C there is some function which provide the facility to allocate and deallocate the memory.

Dynamic memory allocation functions in C :

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

S.No	Function	Syntax
1	malloc()	malloc (number * sizeof(int));
2	calloc()	calloc (number, sizeof(int));
3	realloc()	realloc (pointer_name, number * sizeof(int));
4	free()	free (pointer_name);

malloc() function

- malloc() function is used to allocate space in memory during the execution of the program.
- malloc() does not initialize the memory allocated during execution. It carries garbage value.
- malloc() function returns null pointer if it couldn't able to allocate requested amount of memory.

Example program for malloc() function :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
}
```

```
else
{
    strcpy( mem_allocation,"rahulpublications.com");
}
printf("Dynamically allocated memory content : "\
      "%s\n", mem_allocation );
free(mem_allocation);
}
```

Output :

Dynamically allocated memory content : rahulpublications.com

calloc() function

calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

Example program for calloc() function :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = calloc( 20 * sizeof(char) );
    if( mem_allocation== NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation,"rahulpublications.com");
    }
    printf("Dynamically allocated memory content : "\
          "%s\n", mem_allocation );
    free(mem_allocation);
}
```

Output :

Dynamically allocated memory content : rahulpublications.com

realloc() function

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.

- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

free() function

free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

Example program for realloc() and free() functions :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    char *mem_allocation;
    /* memory is allocated dynamically */
    mem_allocation = malloc( 20 * sizeof(char) );
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation, "rahulpublications.com" );
    }
    printf("Dynamically allocated memory content : "\
        "%s\n", mem_allocation );
    mem_allocation = realloc(mem_allocation, 100 * sizeof(char));
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation, "space is extended upto "\
            "100 characters");
    }
    printf("Resized memory : %s\n", mem_allocation);
    free(mem_allocation);
}
```

OUTPUT :

Dynamically allocated memory content : rahulpublications.com

Resized memory : space is extended upto 100 characters

Static memory allocation Vs Dynamic memory allocation

Static memory allocation	Dynamic memory allocation
1. In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
2. Memory size can't be modified while execution. Example : array	Memory size can be modified while execution. Example : Linked list

malloc() Vs calloc()

malloc()	calloc()
1. It allocates only single block of requested memory.	It allocates multiple blocks of requested memory.
2. <code>int *ptr; ptr = malloc(20 * sizeof(int));</code> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<code>int *ptr; Ptr = calloc(20, 20 * sizeof(int));</code> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3. malloc() doesn't initialize the allocated memory.	calloc() initializes the allocated memory to zero.
4. type cast must be done since this function returns void pointer <code>int *ptr; ptr = (int*)malloc(sizeof(int)*20);</code>	Same as malloc () function <code>int *ptr; ptr = (int*)calloc(20, 20 * sizeof(int));</code>

Short Question & Answers

1. Define function.

Ans :

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

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

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

A function that is built into an application and can be accessed by end-users are called built-in function.

C functions can be classified into two categories :

1. Library functions – printf and scanf are library functions.
2. User-defined functions – main, sqrt, strcat etc. are user-defined functions.

2. List out the advantages of functions. (or) What are the uses of functions?

Ans :

There are many advantages in using functions in a program they are :

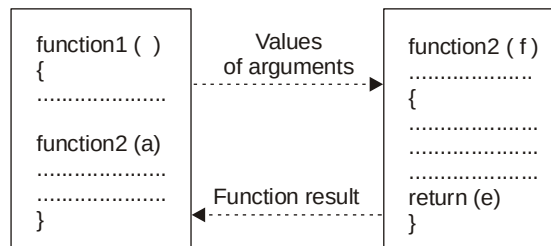
1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
2. The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigation.
4. A function may be used by many other programs this means that a C programmer can build on what others have already done, instead of starting over from scratch.
5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.

3. Explain function with argument and return value.

Ans :

To assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results. These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.

A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such function will have two-way data communication as shown in figure below.



4. Explain about call by value method to pass arguments.

Ans :

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

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

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

5. Explain about call by reference.

Ans :

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

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

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

6. Difference between call-by-value and call by reference.*Ans :*

Call by Value	Call by Reference
1. It is the usual method to call a function in which only the value of the variable is passed as an argument.	In this method, the address of the variable is passed as an argument.
2. Any alteration in the value of the argument or parameter passed is local to the function and is not accepted in the calling program.	Any alteration in the value of the argument or parameter passed is accepted in the calling program (since alteration is made indirectly in the memory location using the pointer).
3. A copy of value is passed to the function	An address of value is passed to the function
4. Memory location occupied by formal and actual arguments is different.	Memory location occupied by formal and actual arguments is same and there is a saving of memory location.
5. Since a new location is created, this method is slow.	Since the existing memory location is used through its address, this method is fast.
6. There is no possibility of wrong data manipulation since the arguments are directly used in an expression.	There is a possibility of wrong data manipulation since the addresses are used in an expression. A good skill of programming is needed here.

7. What are Global Variables ?*Ans :*

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

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main () {
    /* local variable declaration */
    int a, b;
    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

8. What is the use of auto keyword?

Ans :

Automatic variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as an automatic variable unless a different storage-class specification is shown within the declaration.

Since the location of the variable declarations within the program determines the automatic storage class, the keyword `auto` is not required at the beginning of each variable declaration. There is no harm in including an `auto` specification within a declaration, though this is normally not done.

An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

auto is the default storage class for all local variables.

```
{  
    int Count;  
    auto int Month;  
}
```

9. Explain about recursive function. (or) What is recursion?

Ans :

A recursive function is one which calls itself. In C, it is possible for the functions to call themselves. A function is called *recursive* if a statement within the body of a function calls the same function itself.

A very simple example of recursion is presented below :

```
fun()  
{  
    printf("This is an example of recursion\n")  
    fun();  
}
```

When executed, this program will produce an output something like this :

```
This is an example of recursion  
This is an example of recursion  
This is an example of recursion  
This is an example of recursion
```

10. Define pointer. and list out the its benefits.

Ans :

In C a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They are :

1. Pointers are more efficient in handling arrays and 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 passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

11. What is the use of address operator?

Ans :

Address of Operator (&)

The & is a unary operator in C which returns the memory address of the passed operand. This is also known as address of operator.

- Pointer address operator is denoted by '&' symbol
- When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

&n – It gives an address on variable n

Working of Address Operator

```
#include<stdio.h>
void main()
{
    int n = 10;
    printf("\nValue of n is : %d",n);
    printf("\nValue of &n is : %u",&n);
}
```

Output :

Value of n is : 10

Value of &n is : 1002

12. What is the use of NULL Pointer?

Ans :

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

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

```
#include <stdio.h>
int main() {
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

13. Define Dynamic memory allocation.*Ans :*

The process of allocating memory during program execution is called **dynamic memory allocation**. In dynamic memory management memory will be allocated at runtime. This is also known as heap memory. Some language at the run time have ability to calculate and assign the memory space at run time for an array but C can't have this feature. But in C there is some function which provide the facility to allocate and deallocate the memory.

Dynamic memory allocation functions in C :

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

S.No	Function	Syntax
1	malloc()	malloc (number *sizeof(int));
2	calloc()	calloc (number, sizeof(int));
3	realloc()	realloc (pointer_name, number * sizeof(int));
4	free()	free (pointer_name);

14. Differentiate malloc and calloc methods.*Ans :*

malloc()	calloc()
1. It allocates only single block of requested memory.	It allocates multiple blocks of requested memory.
2. <code>int *ptr; ptr = malloc(20 * sizeof(int));</code> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<code>int *ptr;Ptr = calloc(20, 20 * sizeof(int));</code> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3. malloc() doesn't initialize the allocated memory.	calloc() initializes the allocated memory to zero.
4. type cast must be done since this function returns void pointer <code>int *ptr;ptr = (int*)malloc(sizeof(int)*20);</code>	Same as malloc () function <code>int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));</code>

Choose the Correct Answers

1. Use of functions [a, b, c, d]
 - (a) helps to avoid repeating a set of statements many times
 - (b) enhances the logical clarity of the programs
 - (c) helps to avoid repeated programming across programs.
 - (d) makes the debugging task easier
2. Storage class defines _____ [c]
 - (a) the data-type
 - (b) the scope
 - (c) the scope and permanences
 - (d) the scope, permanence and datatype
3. max is a function that returns the larger of the two integers, given as arguments, which of the following statements finds the largest of three given numbers? [a, c, d]
 - (a) max (max (a, b), max (a, c))
 - (b) max (a, max(a, c))
 - (c) max (max(a, b), max(b, c))
 - (d) max (b, max (a, c))
4. c preprocessor [a, b, c, d]
 - (a) takes care of conditional compilation
 - (b) takes care of macros
 - (c) takes care of include files
 - (d) acts before compilation
5. The use of macro in the place of functions [a, d]
 - (a) reduces execution time
 - (b) reduces code size
 - (c) increases execution time
 - (d) increases code size
6. The for loop [a]

```
for (i = 0; i < 10; ++i)
    print f("%d", i & 1)
```

prints

(a) 0101010101	(b) 0111111111
(c) 0000000000	(d) 1111111111

7. The following program [a]
- ```
main ()
{
 int i = 2
 {
 in i = 4, j = 5;
 printf ("%d%d", i, j);
 }
 printf ("%d %d", i, j);
}
```
- (a) will not compile sucessfully (b) Prints 4525  
(c) Prints 2525 (d) none of the above
8. The possible output of printf ("%d %d", ptr, ptr + 1); is [ b ]
- (a) 262 262 (b) 262 266  
(c) 262 263 (d) 262 265
9. In a for loop, if the conditon is missing then, [ b ]
- (a) It is assumed to be present and taken to be false  
(b) It is assumed to be present and taken to be false  
(c) It results in a syntax error.  
(d) execution will be terminated abruptly
10. Using goto inside for loop is equivalent to using [ d ]
- (a) Continue (b) Break  
(c) Return (d) None of the above



## Fill in the Blanks

1. Pointer to pointer can be denoted as \_\_\_\_\_ .
2. The statement that is used to terminate the execution of a function is \_\_\_\_\_ .
3. The type of function depends upon \_\_\_\_\_ .
4. The values given to a function at the time of making the function call are called \_\_\_\_\_.
5. A function that calls itself within its own body is called \_\_\_\_\_.
6. By default, the return type of a function is \_\_\_\_\_
7. The order in which actual arguments are evaluated in a function call is \_\_\_\_\_
8. The address operator and, cannot act on \_\_\_\_\_ .
9. If storage class is missing in the array definition, by default it will be taken to be \_\_\_\_\_.
10. A function that accepts a pointer to a character as argument and returns a pointer to an array of integer can be declared as \_\_\_\_\_.

### ANSWERS

1. \*\*P
2. return statement.
3. Its return type and types of its parameter.
4. actual arguments
5. direct recursive
6. int
7. compiler - dependent
8. arithmetic expressions
9. either automatic or external.
10. int (\*q(char\*)) [ ]

## UNIT IV

**User-defined Data Types :** Declaring a Structure (Union) and its members, Initialization Structure (Union), Accessing members of a Structure (Union), Array of Structures (Union), Structures Vs Unions, Enumeration Types.

**Files in C :** Introduction, Using Files in C, Working with Text Files, Working with Binary Files, Files of Records, Random Access to Files of Records, Other File Management Functions.

### 4.1 USER-DEFINED DATA TYPES

#### 4.1.1 Structures

**Q1. What are structures ? Explain.**

**Ans :** (Nov.-20, June-19, Dec.-17, Imp.)

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types we can use a structure.

Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types.

#### Defining a Structure

Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct tag {
 member 1;
 member 2;

 member m;
};
```

In this declaration, struct is a required keyword; tag is a name that identifies structures of this type (structures having this composition); and member 1, member 2, . . . member *m* are individual member declarations.

The individual members can be ordinary variables, pointers, arrays, or other structures. The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside of the structure. A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure type declaration.

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows :

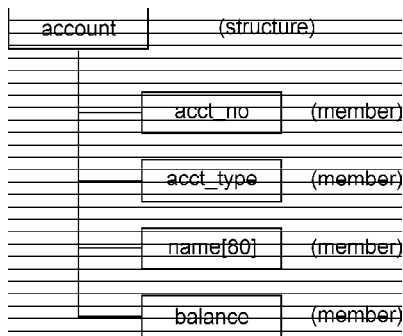
storage-class struct tag variable 1, variable 2, . . . , variable *n* ;

where storage-class is an optimal storage class specifier, struct is a required keyword, tag is the name that appeared in the structure declaration, and variable 1, variable 2, . . . , variable *n* are structure variables of type tag.

A typical structure declaration is shown below.

```
struct account {
 int acct_no;
 char acct_type;
 char name [80];
 float balance;
};
```

This structure is named account (i.e., the tag is account). It contains four members: an integer quantity (acct\_no), a single character (acct\_type), an 80-element character array (name[80]), and floating-point quantity (balance).



#### 4.1.2 Declaring Structure and its Members

#### Q2. Explain declaration, initialization & accessing of structure and its members with example.

**Ans :** (Nov.-20, June-19, Dec.-18, Dec.-16, Imp.)

A structure variable declaration is similar to the declaration of variables of any other data types. It includes the following elements.

1. The keyword **struct**
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

For example, the statement

```
struct book_bank, book1, book2, book3;
```

declares **book1**, **book2**, and **book3** as variables of type **struct book\_bank**.

Each one of these variables has four members as specified by the template. The complete declaration might look like this :

```
struct book_bank
{
 char title[20] ;
 char author[15];
 int pages;
 float price;
};
```

struct book\_bank book1, book2, book3;  
the members of a structure themselves are not

variables. They do not occupy any memory until they are associated with the structure variables such as **book1**. When the compiler comes across a declaration statement, it reserves memory space for the structure variables. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration

```
struct book_bank
{
 char title[20] ;
 char author[15] ;
 int pages;
 float price;
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
struct
{

} book1, book2, book3;
```

declares **book1**, **book2**, and **book3** as structure variables representing three books, but does not include a tag name.

Without a tag name, we cannot use it for future declarations.

Normally, structure definitions appear at the beginning of the program file, before any variables or functions are defined. They may also appear before the **main**, along with macro definitions, such as **#define**. In such cases, the definition is *global* and can be used by other functions as well.

We can use the keyword **typedef** to define a structure as follows :

```
typedef struct
{

 type member1;
 type member2;

} type_name ;
```

The `type_name` represents structure definition associated with it and therefore can be used to declare structure variables as shown below;

```
type_name variable1, variable2, ... ;
```

The name `type_name` is the type definition name, not a variable and we cannot define a variable with typedef declaration.

### Accessing Structure Members

We can access and assign values of the members of a structure in a number of ways. As mentioned earlier, the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word **title**, has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the *member operator* '.' which is also known as 'dot operator' or 'period operator'.

For example,

**book1.price**

is the variable representing the price of **book1** and can be treated like any other ordinary variable.

Here is how we would assign values to the members of **book1** :

```
strcpy(book1.title, " C FUNDAMENTALS");
strcpy(book1.author, "V N Reddy");
book1.pages = 270;
book1.price = 130.90;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s", book1.title);
scanf("%d", &book1.pages);
```

are valid input statements.

The **scanf** and **printf** functions illustrates how the member operator '.' is used to link the structure members to the structure variables is shown in below program. The variable name with a period and the member name is used like an ordinary variable.

```
struct personal
{
 char name[20];
 int day;
 char month[10];
 int year;
 float salary;
};
main()
{
 struct personal person;
 printf("Input Values\n");
 scanf("%s %d %s %d %f", person.name, &person.day,
 person.month, &person.year, &person.salary);
 printf("%s %d %d %f\n", person.name, person.day,
 person.month, person.year, person.salary);
}
```

## Structure Initialization

A structure variable can be initialized at compile time.

```
main()
{
 struct
 {
 int weight;
 float height;
 }
 student = {70, 170.75} ;


```

This assigns the value 70 to **student.weight** and 170.75 to **student.height**. There is one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statement initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
 struct st_record
 {
 int weight ;
 float height;
 } ;
 struct st_record student1 = { 70, 170.75 } ;
 struct st_record student2 = { 64, 180.65 } ;

}
```

Another method is to initialize a structure variable outside the function as shown below :

```
struct st_record
{
 int weight ;
 float height ;
} student1 = {70, 170.75} ;
main()
{
 struct st_record student2 = {64, 180.65} ;

}
```

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables.

The compile-time initialization of a structure variable must have the following elements:

1. The keyword **struct**
2. The structure tag name.
3. The name of the variable to be declared.
4. The assignment operator = .
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

## 4.2 INITIALIZATION OF STRUCTURE

**Q3. Explain different ways to Initialization of structure.**

**Ans :** (Dec.-16)

### C Structure Initialization

When we declare a structure, memory is not allocated for un-initialized variable.

Let us discuss very familiar example of structure student, we can initialize structure variable in different ways

#### Way 1 : Declare and Initialize

```
struct student
{
 char name[20];
 int roll;
 float marks;
} std1 = { "Pritesh",67,78.3 };
```

In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

```
std1 = { "Pritesh",67,78.3 }
```

This is the code for initializing structure variable in C programming

#### Way 2 : Declaring and Initializing Multiple Variables

```
struct student
{
 char name[20];
 int roll;
 float marks;
```

```
}

std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

In this example, we have declared two structure variables in above code. After declaration of variable we have initialized two variable.

```
std1 = {"Pritesh",67,78.3};
std2 = {"Don",62,71.3};
```

#### Way 3 : Initializing Single member

```
struct student
{
 int mark1;
 int mark2;
 int mark3;
} sub1={67};
```

Though there are three members of structure, only one is initialized, Then remaining two members are initialized with Zero. If there are variables of other data type then their initial values will be

| Data Type | Default value if not initialized |
|-----------|----------------------------------|
| integer   | 0                                |
| float     | 0.00                             |
| char      | NULL                             |

#### Way 4 : Initializing inside main

```
struct student
{
 int mark1;
 int mark2;
 int mark3;
};

void main()
{
 struct student s1 = {89,54,65};

};
```

When we declare a structure then memory won't be allocated for the structure. i.e only writing below declaration statement will never allocate memory

```
struct student
```

```
{
 int mark1;
 int mark2;
 int mark3;
};
```

We need to initialize structure variable to allocate some memory to the structure.

```
struct student s1 = {89,54,65};
```

### 4.3 ACCESSING MEMBERS OF A STRUCTURE

**Q4. Explain the process of Accessing Members of a Structure.**

*Ans :*

(Nov.-20, Dec.-16, Imp.)

#### Accessing Structure Members

1. Array elements are accessed using the Subscript variable, Similarly Structure members are accessed using dot [.] operator.
2. (.) is called as "Structure member Operator".
3. Use this Operator in between "Structure name" & "member name"

#### Example

```
#include <stdio.h>
struct Vehicle
{
 int wheels;
 char vname[20];
 char color[10];
} v1 = {4,"Nano","Red"};

int main()
{
 printf("Vehicle No of Wheels : %d",v1.wheels);
 printf("Vehicle Name : %s",v1.vname);
 printf("Vehicle Color : %s",v1.color);
 return(0);
}
```

Output :

Vehicle No of Wheels : 4

Vehicle Name : Nano

Vehicle Color : Red

**Q5. Explain array of structures with example?****Ans :** (June-17)

We use structures to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable. For example :

```
struct class student[100];
```

defines an array called **student**, that consists of 100 elements. Each element is defined to be of the type **struct class**.

Consider the following declaration :

```
struct marks
{
 int subject1;
 int subject2;
 int subject3;
};
main()
{
 struct marks student[3] = { { 45, 68,
81}, {75, 53, 69}, {57, 36, 71} } ;
```

This declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows :

```
student[0].subject1 = 45;
student[0].subject2 = 65;
. . . .
. . . .
student[2].subject3 = 71;
```

Here, the array is declared just as it would have been with any other array. Since student is an array, we use the usual array-accessing methods to access individual elements and then the member operator to access members. Each element of student array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array student actually looks as shown in figure below.

|                      |    |
|----------------------|----|
| student[0].subject 1 | 45 |
| .subject 2           | 68 |
| .subject 3           | 81 |
| student[1].subject 1 | 75 |
| .subject 2           | 53 |
| .subject 3           | 69 |
| student[2].subject 1 | 57 |
| .subject 2           | 36 |
| .subject 3           | 71 |

**Fig.: The array student inside memory**

**Program to calculate the subject-wise and student-wise totals and store them as a part of the structure.**

```
struct marks
{
 int sub1;
 int sub2;
 int sub3;
 int total;
};
main()
{
 int i ;
 struct marks student[3] = { {45, 67, 81, 0},
{75, 53, 69, 0}, {57, 36, 71, 0} } ;
 struct marks total ;
 for(i = 0; i <= 2; i++)
 {
```



```

 student[i].total = student[i].sub1 + student[i].sub2 + student[i].sub3;
 total.sub1 = total.sub1 + student[i].sub1;
 total.sub2 = total.sub2 + student[i].sub2;
 total.sub3 = total.sub3 + student[i].sub3;
 }
 printf(" STUDENT TOTAL\n\n") ;
 for(i = 0; i <= 2; i++)
 printf("Student[%d] %d\n", i+1,student[i].total) ;
 printf("\n SUBJECT TOTAL\n\n");
 printf("%s %d\n%s %d\n%s %d\n",
 "Subject 1 ", total.sub1,
 "Subject 2 ", total.sub2,
 "Subject 3 ", total.sub3);
 printf("\nGrand Total = %d\n", total.total) ;
}

```

**Output :**

| STUDENT           | TOTAL |
|-------------------|-------|
| Student [1]       | 193   |
| Student [2]       | 197   |
| Student [3]       | 164   |
| SUBJECT           | TOTAL |
| Subject 1         | 177   |
| Subject 2         | 156   |
| Subject 3         | 221   |
| Grand Total = 554 |       |

**Q6. What are nested structures ? Explain with example.**

**Ans :**

Structures within a structure means nesting of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```

struct salary
{
 char name;
 char department;
 int basic_pay;
 int dearness_allowance;
 int house_rent_allowance;
 int city_allowance;
}
employee;

```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below :

```
struct salary
{
 char name;
 char department;
 struct
 {
 int dearness;
 int house_rent;
 }
 allowance ;
}
employee;
```

The salary structure contains a member named allowances which itself is a structure with three members. The members contained in the inner structure namely dearness, house\_rent, and city can be referred to as:

```
employee.allowance.dearness
employee.allowance.house_rent
employee.allowance.city
```

An inner-most member in a nested structure can be accessed by chaining all the concerned structure variables (from outer-most to inner-most) with the member using dot operator. The following are invalid :

```
employee.allowance(actual member is missing)
employee.house_rent(inner structure variable is missing)
```

An inner structure can have more than one variable. The following form of declaration is legal :

```
struct salary
{

 struct
 {
 int dearness;

 }
 allowance,
 arrears;
}
employee[100];
```

The inner structure has two variables, allowance and arrears. This implies that both of them have the same structure template. Note the comma after the name allowance. A base member can be accessed as follows :

```
employee[1].allowance.dearness
employee[1].arrears.dearness
```

We can also use tag names to define inner structures.

### Example

Use of tag names to define inner structures

```
struct pay
{
 int dearness;
 int house_rent;
 int city;
};
struct salary
{
 char name ;
 char department ;
 struct pay allowance ;
 struct pay arrears;
};
struct salary employee[100] ;
```

pay template is defined outside the salary template and is used to define the structure of allowance and arrears inside the salary structure.

It is also permissible to nest more than one type of structures.

```
struct personal_record
{
 struct name_part name;
 struct addr_part address;
 struct date date_of_birth;

};
struct personal_record personal ;
```

The first member of this structure is name, which is of the type struct name\_part. Similarly, other members have their structure types.

**Q7. What are the differences between structures and arrays.****Ans :**

| Structures                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                | Arrays                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>A structure is a collection of variables of heterogeneous data types.</p> <p>A structure is declared by using the keyword struct as follows :</p> <pre>struct structure_name {     type1field1;     type2field2;     :     typen fieldn; }</pre> <p>The individual members of the structure are accessed using the . (dot) operator.</p> <pre>Struct_variable.field_name</pre> <p><b>Rules to be followed while using structures</b></p> <p>Define structure</p> <p>Declare the structure variables to be used in program.</p> <p>Use '.' (dot) operator to access the members of the ordinary structure variable.</p> <p>Use * operator to access the members of the pointer structure variables.</p> | <p>An array is a collection of variables of homogeneous types.</p> <p>An array can declare without using any keyword but just by specifying the data type that must be followed by each element of the array and the maximum dimension of the array. This can be given as</p> <pre>datatype_arr_name[max_dimension];</pre> <p>The individual members of the array are accessed using the subscript. The number specified within the subscript specifies the element to be accessed. The maximum subscript value is max_dimension-1. The minimum subscript value is 0.</p> <p><b>Rules to be followed while using arrays</b></p> <p>Declare an array</p> <p>During declaration mention sizes of the dimensions</p> <p>To access the array elements use subscript.</p> <p>Do not access beyond the size of the array. It may give incorrect results.</p> <p>All the elements of an array must be of same data type.</p> |

**Q8. Explain structures with functions?****Ans :**

C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.
2. Second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the functions to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

- The third approach employs a concept called *pointers* to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function.

The general format of sending a copy of a structure to the called function is :

```
function_name (structure_variable_name);
```

The called function takes the following form :

```
data_type function_name (struct_type st_name)
{

 return(expression) ;
}
```

- The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, than it must be declared as struct with an appropriate tag name.
- The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
- The return statement is necessary only when the function is returning some data back to the calling function. The *expression* may be any simple variable or structure variable or an expression using simple variables.
- When a function returns a structure, it must be assigned to a structure of identical type in the calling function.

**Program to illustrate the method of sending an entire structure as a parameter to a function.**

```
/* Passing a copy of the entire structure */
struct stores
{
 char name[20] ;
 float price ;
 int quantity ;
};
struct stores update (struct stores product, float p, int q) ;
float mul (struct stores stock) ;
main()
{
 float p_increment, value ;
 int q_increment ;
 struct stores item = {"XYZ", 25.75, 12} ;
 printf("\n Input increment values :") ;
 printf(" price increment and quantity increment \n") ;
 scanf("%f %d", &p_increment, &q_increment) ;
/* */
 item = update(item, p_increment, q_increment) ;
/* */
```

```

 printf("Updated values of item\n\n");
 printf("Name : %s\n", item.name);
 printf("Price : %f\n", item.price);
 printf("Quantity : %d\n", item.quantity);
 /* */
 value = mul (item);
 /* */
 printf("\nValue of the item = %f\n", value);
}
struct stores update(struct stores product, float p, int q)
{
 product.price += p;
 product.quantity += q;
 return(product);
}
float mul (struct stores stock)
{
 return(stock.price * stock.quantity);
}

```

**Output :**

```

Input increment values : price increment and quantity increment
10 12
Updated values of item
Name : XYZ
Price : 35.750000
Quantity : 24
Value of the item = 858.000000

```

The function update receives a copy of the structure variable item as one of its parameters. Note that both the function update and the formal parameter product are declared as type struct stores. It is done so because the function uses the parameter product to receive the structure variable item is declared as type struct stores.

The entire structure returned by update can be copied into a structure of identical type. The statement

```
item = update(item,p_increment, q_increment);
```

replaces the old values of item by the new ones.

## 4.4 UNIONS

### Q9. What are unions ? Explain.

Ans :

A union, is a collection of variables of different types, just like a structure. However, with unions, you can only store information in one field at any one time.

A union can also be viewed as a variable type that can contain many different variables, but only actually holds one of them at a time (not like a structure). This can save memory if you have a group of data where only one of the types is used at a time.

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computers memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to observe memory. They are useful for application involving multiple members. Where values need not be assigned to all the members at any one time.

### Usage

- access individual bytes of larger type
- variable format input records (coded records)
- sharing an area to save storage usage
- unions not used nearly as much as structures

Like structures union can be declared using the keyword union as follows:

```
union item
{
 int m;
 float p;
 char c;
}
code;
```

this declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is because if only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member we can use the same syntax that we use to access structure members. That is

```
code.m
code.p
code.c
```

are all valid member variables. During accessing we should make sure that we are accessing the member whose value is currently stored.

For example a statement such as

```
code.m=456;
code.p=456.78;
printf("%d",code.m);
```

would produce erroneous result.

### Program to illustrate the concept of union

```
#include<stdio.h>
main()
{
 union student
 {
 char grade;
 int rollno;
 float marks;
 double fees;
 }
 union student s;
 printf("\n Enter the Grade, Rollno, Marks
and Fees of the Student : ")
 scanf("\n %c%d%f%g", &s.grade,
 &s.rollno, &s.marks, &s.fees);
 printf("\n Grade is : %c", s.grade);
 printf("\n Rollno is : %d", s.rollno);
 printf("\n Marks are : %f", s.marks);
 printf("\n Fees paid : %g", s.fees);
 getch();
}
```

## 4.5 STRUCTURES Vs UNION

### Q10. What are the differences between structures & unions ?

**Ans :** (June-18, Dec.-16, Imp.)

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a *member* or *field*.

A *union* is an object similar to a structure except that all of its members start at the same location in memory. A union variable can represent the value of only one of its members at a time.

Similarities between Structures and Unions :

**Structures and unions** share the following characteristics:

- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- The only operators valid for use with entire structures and unions are the simple assignment (=) and sizeof operators. In particular, structures and unions cannot appear as operands of the equality (==), inequality (!=), or cast operators. The two structures or unions in the assignment must have the same members and member types.
- A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.
- Several members of a structure can be initialized at once; only the first member of a union can be given an initializer.
- In structures, every member has its own memory where as in union, all members use the same memory.
- Different interpretations of the same memory location are not possible in structures. In union, different interpretations of the same memory location are possible.

#### 4.6 ENUMERATION TYPES

#### Q11. What are enumerated data types?

Ans :

(Dec.-18)

An enumeration is a data type, similar to a structure or a union. Its members are constants that are written as identifiers, though they have signed integer values. These constants represent values that can be assigned to corresponding enumeration variables.

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the “enumeration set,” “enumerator constants,” “enumerators,” or “members”). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

Variables of **enum** type can be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the **#define** preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

#### Syntax

*enum-specifier* :

#### The difference between structures and unions lies in the way their members are stored and initialized, as follows :

- Within a structure, the members have addresses that increase as the declarators are read left-to-right. That is, the members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0.  
A pointer to a structure points to its first member, so no unnamed holes can reside at the beginning of a structure.
- In a union, every member begins at offset 0 from the address of the union. The size of the union in memory is the size of its largest member. The value of only one member can be stored in a union object at a time. When the storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. The rules for alignment of union members are the same as for structure members.  
A pointer to a union member, converted to the proper type, points to the beginning of the union object.

```
enum identifieropt { enumerator-list }
enum identifier
```

The optional *identifier* names the enumeration type defined by *enumerator-list*. This identifier is often called the “tag” of the enumeration specified by the list. A type specifier of the form

```
enum identifier { enumerator-list }
```

declares *identifier* to be the tag of the enumeration specified by the *enumerator-list* nonterminal. The *enumerator-list* defines the “enumerator content.” The *enumerator-list* is described in detail below.

If the declaration of a tag is visible, subsequent declarations that use the tag but omit *enumerator-list* specify the previously declared enumerated type. The tag must refer to a defined enumeration type, and that enumeration type must be in current scope. Since the enumeration type is defined elsewhere, the *enumerator-list* does not appear in this declaration. Declarations of types derived from enumerations and **typedef** declarations for enumeration types can use the enumeration tag before the enumeration type is defined.

Below examples illustrate enumeration declarations :

```
enum DAY /* Defines an enumeration type */
{
 saturday, /* Names day and declares a */
 sunday = 0, /* variable named workday with */
 monday, /* that type */
 tuesday,
 wednesday, /* wednesday is associated with 3 */
 thursday,
 friday
} workday;
```

The value 0 is associated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In this example, a value from the set *DAY* is assigned to the variable *today*.

```
enum DAY today = wednesday;
```

The name of the enumeration constant is used to assign the value. Since the *DAY* enumeration type was previously declared, only the enumeration tag *DAY* is necessary.

### Advantages of using the enumerations in program

1. Enumeration constants are automatically assigned equivalent integer values , beginning with 0 for the first constant, with each successive constant increasing by 1, and it can be overridden the some other values.
2. Use of enumeration variables within a program can often increase the logical clarity of that program
3. Enumeration variables are particularly used as flags to indicate various options for carrying out a calculation.



The following rules apply to the members of an enumeration set :

1. An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers, perhaps named null and zero, in the same set.
2. The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
3. Enumeration tags obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.

#### 4.7 FILES IN C

**Q12. What are files ? Explain its types ?**

**Ans :**

**(June-19, Dec.-17)**

File is a set of records that can be accessed through a set of library functions. These functions are available in `stdio.h`. A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level (OS level) calls to handle file on your storage devices.

#### Streams and File Types

Reading and writing of data is done with streams. The streams are designed to allow the user to access the files efficiently. A stream is a file and using a physical device like keyboard, information is stored in the file. Printer and monitor are used to display stream. The `FILE` object uses these devices.

The `FILE` object contains all the information about a stream like its current position, pointer to any buffer, error, and EOF (end of file). Using this information of object, C program uses pointer, which is returned from stream function `fopen()`. A function `fopen()` is used for opening a file.

#### File Types

There are two types of files :

1. Sequential File and
  2. Random Accesses File.
1. **Sequential File :** In this type data are kept sequentially. If we want to read the last record of the file we need to read all the records before that particular record. It takes more time to access the records. Or if we desire to access the 10th record, then the first 9 records should be read sequentially for reaching the 10th record.
  2. **Random Access File :** In this type data can be read and modified randomly. That is, if we want to read the last record of the file, we can read it directly. It takes less time as compared to a sequential file.

#### Steps for File Operations

There are three steps for file operation in C.

1. Opening a file
2. Reading or writing a file
3. Closing a file

Most of the compilers support file handling functions. C language also supports many file handling functions which are available in standard library and are listed below. All the functions that support C are not listed here because all compilers do not support all the functions.

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

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

---

**Q13. What is the significant of EOF?**

Ans :

End-of-file (commonly abbreviated EOF) is a condition in a computer operating system where no more data can be read from a data source. The data source is usually called a file or stream.

**Syntax**

```
int feof(FILE *stream);
```

EOF does –

- Macro tests if end-of-file has been reached on a stream.
- feof is a macro that tests the given stream for an end-of-file indicator.
- Once the indicator is set, read operations on the file return the indicator until rewind is called, or the file is closed.
- The end-of-file indicator is reset with each input operation.

**Detecting End of File****In Text File**

- Special Character EOF denotes the end of File
- As soon as Character is read, End of the File can be detected
- EOF is defined in stdio.h
- Equivalent value of EOF is -1

**Printing Value of EOF**

```
void main()
{
 printf("%d", EOF);
}
```

**In Binary File**

- feof function is used to detect the end of file
- It can be used in text file
- feof Returns TRUE if end of file is reached

**Syntax**

```
int feof(FILE *fp);
```

**Writing feof Function**

In if statement

```
if(feof(fp) == 1) // as if(1) is TRUE
 printf("End of File");
```

In While loop

```
while(!feof(fp))
{

}
```

**Q14. Explain essential actions required to access the file.**

*Ans :*

A file has to be opened before beginning of read and write operations. Opening a file creates a link between the operating system and the file functions. We have to specify the name of file and its mode to the operating system. This important task is carried out by the structure FILE that is defined in stdio.h header file. So this file must be included.

When a request is made to the operating system for opening a file, it does so by granting the request. If request is granted, the operating system points to the structure FILE. In case the request is not granted, it returns NULL. That is why the following declaration before opening of the file is to be made.

**FILE \*fp**

where, **fp** is file pointer.

Each file that we open has its own FILE structure. The information that the file contains may be its current size, its location in memory, and so on. The only one function to open a file is **fopen( )**.

**Syntax**

```
FILE *fp;
fp = fopen("data.txt", "r");
```

Here, **fp** is a pointer variable that contains address of the structure FILE that has been defined in the header file "**stdio.h**". The function **fopen( )** will open a file "**data.txt**" in read mode. The C compiler reads the contents of the file because it finds the read mode ("**r**"). Here, "**r**" is a string and not a character. Hence, it is enclosed with double quotes and not with single quotes.

The **fopen( )** performs these important tasks.

1. It searches the disk for opening the file.
2. In case the file exists, it loads the file from the disk into memory. If the file is found with huge contents then it loads the file part by part.
3. If the file does not exist, this function returns a NULL. NULL is a macro-defined character in the header file "**stdio.h**". This indicates that it is unable to open the file.

There may be following reasons for the failure of **fopen( )** functions : (a) when the file is in protected or hidden mode, (b) the file may be used by another program.

4. It locates a character pointer, which points the first character of the file. Whenever a file is opened the character pointer points to the first character of the file.

Table below are different modes in which a file can be opened.

| Mode | Description                                                                                                                                                                     |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| r    | Opens an existing text file for reading purpose.                                                                                                                                |
| w    | Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.                   |
| a    | Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content. |
| r +  | Opens a text file for both reading and writing.                                                                                                                                 |
| w +  | Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.                         |
| a +  | Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.           |
| rb   | opens a binary file in reading mode                                                                                                                                             |
| wb   | opens or create a binary file in writing mode                                                                                                                                   |
| ab   | opens a binary file in append mode                                                                                                                                              |
| rb + | opens a binary file in both reading and writing mode                                                                                                                            |
| wb + | opens a binary file in both reading and writing mode                                                                                                                            |
| ab + | opens a binary file in both reading and writing mode                                                                                                                            |

#### Q15. Explain the methods used for accessing a file?

Ans :

To close a file, use the `fclose()` function. The prototype of this function is -

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

The **fclose()** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

#### Writing a File

Following is the simplest function to write individual characters to a stream-

```
int fputc(int c, FILE *fp);
```

The function **fputc()** writes the character value of the argument `c` to the output stream referenced by `fp`. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream -

```
int fputs(const char *s, FILE *fp);
```

The function **fputs()** writes the string `s` to the output stream referenced by `fp`. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE \*fp, const char \*format, ...)** function as well to write a string into a file.

**Example**

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine.

```
#include <stdio.h>
main() {
 FILE *fp;
 fp = fopen("/tmp/test.txt", "w+");
 fprintf(fp, "This is testing for fprintf...\n");
 fputs("This is testing for fputs...\n", fp);
 fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file **test.txt** in **/tmp** directory and writes two lines using two different functions.

**Reading a File**

Given below is the simplest function to read a single character from a file -

```
int fgetc(FILE *fp);
```

The **fgetc()** function reads a character from the input file referenced by **fp**. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream -

```
char *fgets(char *buf, int n, FILE *fp);
```

The functions **fgets()** reads up to **n - 1** characters from the input stream referenced by **fp**. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character.

You can also use **int fscanf(FILE \*fp, const char \*format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

```
#include <stdio.h>
main() {
 FILE *fp;
 char buff[255];
 fp = fopen("/tmp/test.txt", "r");
```

```
fscanf(fp, "%s", buff);
printf("1 : %s\n", buff);
fgets(buff, 255, (FILE*)fp);
printf("2: %s\n", buff);
fgets(buff, 255, (FILE*)fp);
printf("3: %s\n", buff);
fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result :

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

**4.8 USING FILES IN C****Q16. Give a basic introduction about Files in C ?**

**Ans :** **(Nov.-20, June-18)**

**File**

It is necessary to keep data in the permanent storage because it is difficult to handle the large volume of data by programs and after execution of program is over all the entered data will be lost because, the data stored in the variables are temporary.

C supports the concept of file through which the data can be stored in the disk or secondary storage device. So the data can be read when required. A file is a collection of data or text, placed on the disk.

There are two types of files- sequential file and random access file. In sequential file, data are kept sequential. As example if we want to access

the forty fourth record, then first forty three records should be read sequentially to reach the forty fourth record. In the record access file, the data can be accessed and processed randomly i.e in this case the forty fourth record can be accessed directly. It takes less time than the sequential file.

Hence depending up on the method of accessing the data stored ,there are two types of files.

1. Sequential file
2. Random access file

### 1. Sequential File

In this type of files data is kept in sequential order if we want to read the last record of the file,we need to read all records before that record so it takes more time.

### 2. Random access Files

In this type of files data can be read and modified randomly .If we want to read the last record we can read it directly.It takes less time when compared to sequential file.

The steps for file operation in C programming are as follows:

1. Open a file.
2. Read the file or write data in the file.
3. Close the file.

### Q17. List out basic File operation ?

**Ans :** (Dec.-19, June-18)

#### File

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

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

FILE \*fp;

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

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

### Q18. Explain about Opening ,Reading, Writing and closing a file ?

**Ans :** (June-18, June-17, Imp.)

#### Opening or creating file

For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below.

#### File opening modes in C:

- **"r"** – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
- **"w"** – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"a"** – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.
- **"r+"** – Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
- **"w+"** – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.

- **"a+"** – Searches file. If the file is opened successfully `fopen( )` loads it into memory and sets up a pointer which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

As given above, if you want to perform operations on a binary file, then you have to append 'b' at the last. For example, instead of "w", you have to use "wb", instead of "a+" you have to use "a+b". For performing the operations on the file, a special pointer called File pointer is used which is declared as

```
FILE *filePointer;
```

So, the file can be opened as

```
filePointer = fopen("fileName.txt", "w")
```

The second parameter can be changed to contain all the attributes listed in the above table.

- **Reading from a file**

The file read operations can be performed using functions `fscanf` or `fgets`. Both the functions performed the same operations as that of `printf` and `gets` but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.

And the code snippet for reading a file is as:

```
FILE * filePointer;
```

```
filePointer = fopen("fileName.txt", "r");
```

```
fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);
```

- **Writing a file**

The file write operations can be performed by the functions `fprintf` and `fputs` with similarities to read operations. The snippet for writing to a file is as :

```
FILE *filePointer ;
```

```
filePointer = fopen("fileName.txt", "w");
```

```
fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);
```

- **Closing a file**

After every successful file operations, you must always close a file. For closing a file, you have to use `fclose` function. The snippet for closing a file is given as :

- `FILE *filePointer ;`

- `filePointer = fopen("fileName.txt", "w");`

- `_____ Some file Operations _____`

```
fclose(filePointer)
```

## 4.9 WORKING WITH TEXT FILES

**Q19. Explain the functions used in reading and writing of text files ?**

Ans :

(Dec.-19, Dec.-18, Dec.-16, Imp.)

In the above table we have discussed about various file I/O functions to perform reading and writing on file. `getc()` and `putc()` are simplest functions used to read and write individual characters to a file. C provides four functions that can be used to read text files from the disk. These are

- `fscanf()`
- `fgets()`
- `fgetc()`
- `fread()`

C provides four functions that can be used to write text files into the disk. These are

- fprintf()
- fputs()
- fputc()
- fwrite()

### Character Input and Output

When used with disk files, the term *character I/O* refers to single characters as well as lines of characters since a line is nothing but a sequence of zero or more characters terminated by the new-line character. Character I/O is used with textmode files. The following sections describe character input/output functions for files with suitable examples.

**putc() Function :** The library function `putc()` writes a single character to a specified stream. Its prototype in `stdio.h` appears as follows

```
int putc(int ch, FILE *fp);
```

The argument `ch` is the character to be outputted. As with other character functions, it is formally considered to be of type `int`, but only the lower-order byte is used. The argument `fp` is the pointer associated with the file, which is the pointer returned by `fopen()` when the file is opened. The function `putc()` returns the character just written if successful or EOF if an error occurs. The symbolic constant EOF is defined in `stdio.h`, and it has the value `-1`. Because no 'real' character has that numeric value, EOF can be used as an error indicator with text-mode files only.

The following program illustrates how to write a single character at a time into a text file.

```
#include <stdio.h>
int main()
{
 FILE *fp;
 char text[80];
 int i, c;
 fp = fopen("abc.txt", "w"); /* Here abc
is file name, w is write mode */
 printf("\n ENTER TEXT");
 scanf("%[^\n]", text);
 for(c = 1; c <= 10; c++)
 {
 for(i = 0; text[i]; i++)
```

```
 putc(text[i], fp);
 putc('\n', fp);
 }
 fclose(fp);
 return 0;
}
```

To append more lines to the file `abc.txt`, the statement in bold font has to be replaced with the statement `fp = fopen("abc.txt", "a");`

### getc() and fgetc() functions

The functions `getc()` and `fgetc()` are identical and can be used interchangeably. They input a single character from the specified stream. The following is the prototype of `getc()` in `stdio.h`.

```
int getc(FILE *fp);
```

The argument `fp` is the pointer returned by `fopen()` when the file is opened. `getc()` reads a single character from the opened file and moves the file pointer. It returns EOF, if end of file is reached.

`fgetc()` is similar to `getc()` function. It also reads a character and increases the file pointer position. If any error or end of file is reached it returns EOF.

### fprint() and fscanf()

**fprint()** function is used for writing characters, strings, integers, floats, and so on, to the file. It contains one more parameter that is file pointer, which points the opened file. Here is a program that illustrates the use of **fprint()** function.

**Program to open a text file and write some text using fprintf() function. Open the file and verify the contents.**

```
#include <stdio.h>
void main()
{
 FILE *fp;
 char text[30];
 fp = fopen("Text.txt", "w");
 clrscr();
 printf("Enter Text Here :");
 gets(text);
 fprintf(fp, "%s", text);
 fclose(fp);
}
```

Output :

Enter Text Here : Have a nice day.



In the above program, `fprintf( )` functions writes the string to the file pointed by `fp`. The string is collected through `gets( )` function into character array `name[30]`.

**fscanf( )** function reads characters, strings, integers, floats, and so on, from the file pointed by file pointer. A program is illustrated below based on this.

**Program to enter data into the text file and read the same. Use “w+” file mode. Use `fscanf( )` to read the contents of the file.**

```
#include <stdio.h>
void main()
{
 FILE *fp;
 char text[15];
 int age;
 fp=fopen("Text.txt", "w+");
 clrscr();
 printf("Name\tAGE\n");
 scanf("%s %d", text, &age);
 fprintf(fp, "%s %d", text, age);
 printf("Name\tAGE\n");
 fscanf(fp, "%s %d", text, &age);
 printf("%s\t%d\n", text, age);
 fclose(fp);
}
```

Output :

|         |     |
|---------|-----|
| Name    | AGE |
| RAHUL   | 15  |
| Name    | AGE |
| NARAYAN | 18  |

In the above program, **fscanf( )** reads the data from the file named “text.txt”.

#### 4.10 WORKING WITH BINARY FILES

**Q20. Explain the working of binary files ?**

**Ans :** (Nov.-20, June-19, Dec.-17)

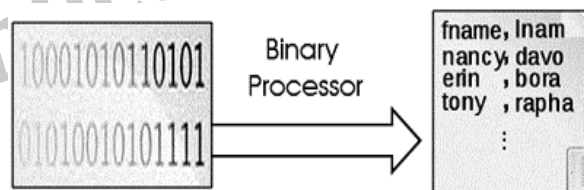
Binary files are very similar to arrays of structures, except the structures are in a disk-file rather than an array in memory. Binary files have two features that distinguish them from text files:

- You can instantly use any structure in the file.
- You can change the contents of a structure anywhere in the file.

After you have opened the binary file, you can read and write a structure or seek a specific position in the file. A file position indicator points to record 0 when the file is opened.

#### What are Binary Files

- Binary Files Contain Information Coded Mostly in Binary Format.
- Binary Files are difficult to read for human.
- Binary Files can be processed by certain applications or processors.
- Only Binary File Processors can understand Complex Formatting Information Stored in Binary Format.
- Humans can read binary files only after processing.
- All Executable Files are Binary Files.



As shown in fig. Binary file is stored in Binary Format (in 0/1). This Binary file is difficult to read for humans. So generally Binary file is given as input to the Binary file Processor. Processor will convert binary file into equivalent readable file.

Some Examples of the Binary files :

1. Executable Files
2. Database files

#### 4.11 RANDOM ACCESS TO FILES OF RECORDS

**Q21. Explain the functions to access Random files in C ?**

**Ans :** (June-18, June-17)

Random access means we can move to any part of a file and read or write data from it without having to read through the entire file. we can access the data stored in the file in two ways. They are sequentially and randomly.

if we want to access the forty fourth record then first forty three record read sequentially to reach forty four record .In random access data can be accessed and processed directly .There is no need to read each record sequentially .if we want to access a particular record random access takes less time than the sequential access.

C supports these function for random access file.

1. fseek( ) Function
2. ftell ( ) Function
3. rewind() Function

### 1. fseek():

This function is used for seeking the pointer position in the file at the specified byte.

**Syntax:** fseek( file pointer, displacement, pointer position);

Where

**file pointer** — It is the pointer which points to the file.

**displacement** — It is positive or negative. This is the number of bytes which are skipped backward (if negative) or forward (if positive) from the current position. This is attached with L because this is a long integer.

### Pointer position:

This sets the pointer position in the file.

| Value | pointer position   |
|-------|--------------------|
| 0     | Beginning of file. |
| 1     | Current position   |
| 2     | End of file        |

### Ex:

1. fseek( p,10L,0)

0 means pointer position is on beginning of the file, from this statement pointer position is skipped 10 bytes from the beginning of the file.

- 2.) fseek( p,5L,1)

1 means current position of the pointer position. From this statement pointer position is skipped 5 bytes forward from the current position.

3. fseek(p,-5L,1)

From this statement pointer position is skipped 5 bytes backward from the current position.

### 2. ftell()

This function returns the value of the current pointer position in the file. The value is count from the beginning of the file.

**Syntax:** ftell(fp);

Where fp is a file pointer.

### 3. rewind()

This function is used to move the file pointer to the beginning of the given file.

**Syntax:** rewind(fp);

Where fp is a file pointer.

#### 4.12 OTHER FILE MANAGEMENT FUNCTIONS

##### Q22. Explain about file management functions and operations ?

Ans : (Dec.-18, June-18, June-17)

Using C you can write programs that works as common DOS commands like deleting, renaming and copying the disk files. The copy and delete operations are also associated with file management. Though one could write programs for them, the C standard library contains functions for deleting and renaming files.

##### Deleting a File

To delete a file in c, We use the **remove** macro, which is defined in `stdio.h`. `remove` macro takes filename with extension as its argument and deletes the file, if it is in the directory and returns the zero, if deleted successfully. Note that deleted file does not goes to the recycle bin, it is going to delete permanently.

The C library function **int remove(const char \*filename)** deletes the given **filename** so that it is no longer accessible.

##### Declaration

Following is the declaration for `remove()` function.

```
int remove(const char *filename)
```

**filename** – This is the C string containing the name of the file to be deleted. On success, zero is returned. On error, -1 is returned, and `errno` is set appropriately.

##### Example

The following example shows the usage of `remove()` function.

```
#include<stdio.h>
#include<string.h>
int main()
{
 int ret;
 FILE *fp;
 char filename[] = "file.txt";
```

```
fp = fopen(filename, "w");
fprintf(fp, "%s", "This is C Language");
fclose(fp);
ret = remove(filename);
if(ret == 0)
{
 printf("File deleted successfully");
}
else
{
 printf("Error: unable to delete the file");
}
return(0);
}
```

Let us assume we have a text file `file.txt` having some content. So we are going to delete this file, using the above program. Let us compile and run the above program to produce the following message and the file will be deleted permanently.

File deleted successfully

##### Renaming a File

The `rename()` function changes the name of an existing disk file. The C library function **int rename(const char\*old\_filename, const char \*new\_filename)** causes the filename referred to by **old\_filename** to be changed to **new\_filename**.

##### Declaration

Following is the declaration for `rename()` function.

```
int rename(const char *old_filename,
const char *new_filename)
```

##### Parameters

- **Old\_filename** : This is the C string containing the name of the file to be renamed and/or moved.
- **New\_filename** : This is the C string containing the new name for the file.

On success, zero is returned. On error, -1 is

returned, and `errno` is set appropriately.

### Example

The following example shows the usage of `rename()` function.

```
#include <stdio.h>
int main()
{
 int ret;
 char oldname[] = "file.txt";
 char newname[] = "newfile.txt";
 ret = rename(oldname, newname);
 if(ret == 0)
 {
 printf("File renamed successfully");
 }
 else
 {
 printf("Error: unable to rename the file");
 }
 return(0);
}
```

Let us assume we have a text file **file.txt**, having some content. So, we are going to rename this file, using the above program. Let us compile and run the above program to produce the following message and the file will be renamed to **newfile.txt** file.

File renamed successfully

### Copying a File

It's frequently necessary to make a copy of a file or duplicate with a different name or with the same name but in a different drive or directory. In DOS, you can do this with the **COPY** command, and in UNIX using **cp**. But in C there's no library function, so you need to write your own program. Here are the steps you need follow :

1. Open the source file for reading in binary mode. (Using binary mode ensures that the function can copy all sorts of files, not just text files.)

2. Open the destination file for writing in binary mode.
3. Read a character from the source file. Remember, when a file is first opened, the pointer is at the start of the file, so there's no need to position the file pointer explicitly.
4. If the function `feof( )` indicates that you've reached the end of the source file, you're done and can close both files and return to the calling program.
5. If you haven't reached end-of-file, write the character to the destination file, and then loop back to step 3.

### Example Program

```
#include<stdio.h>
int copy_file(char *old_filename, char
*new_filename)
{
 FILE *ptr_old, *ptr_new;
 errno_t err = 0, err1 = 0;
 int a;
 err = fopen_s(&ptr_old, old_filename,
"rb");
 err1 = fopen_s(&ptr_new,
new_filename, "wb");
 if(err != 0)
 return -1;
 if(err1 != 0)
 {
 fclose(ptr_old);
 return -1;
 }
 while(1)
 {
 a = fgetc(ptr_old);
 if(!feof(ptr_old))
 fputc(a, ptr_new);
 else
 break;
 }
 fclose(ptr_new);
 fclose(ptr_old);
 return 0;
}
int main(void)
{
```

```
char filename_src[101], filename_dest[101];
printf("\nSource file: ");
gets_s(filename_src, 100);

printf("\nDestination filename: ");

gets_s(filename_dest, 100);

if(copy_file(filename_src, filename_dest) == 0)

 printf("Copy Successful\n");

else

 fprintf(stderr, "Error during copy!");

}
```

First we take a look at main before we look at the function copy\_file().

### **main()**

In main() we first make two buffers, a source and destination buffer, that each can hold 100 characters plus '\0'. Using the gets\_s function (a more secure gets) the program waits for input from the user. The user must input a source file and a destination file. In the "if statement" the function copy\_file() is called. The function is either successful or will fail. A message is printed accordingly.

### **copy\_file()**

In main the function copy\_file() is called with the source filename and the destination file. The two files are then opened. The first file is opened in read only binary mode. The second file is opened in write binary mode. (If the file doesn't exist the file is created.) For opening the file fopen\_s() is used which is a more secure fopen().

After opening some error checking is done. If one of the two fopen\_s() functions fails an -1 is returned.

In the "while statement" the content of the source file is read one byte at the time. This byte is written to the destination file. This will continue until the end of the file (EOF) is reached. If the end of the file is reached we break out with a break command.

Before the program ends both files are closed using fclose() function.

## Short Question & Answers

### 1. Define Structure.

Ans :

Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type. If we need to use a collection of different data type items it is not possible using an array. When we require using a collection of different data items of different data types we can use a structure.

Structure is a method of packing data of different types. A structure is a convenient method of handling a group of related data items of different data types.

### Defining a Structure

Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct tag {
 member 1;
 member 2;

 member m;
};
```

### 2. Explain process of initialization a structure.

Ans :

#### C Structure Initialization

When we declare a structure, memory is not allocated for un-initialized variable.

Let us discuss very familiar example of structure student , we can initialize structure variable in different ways

#### Way 1 : Declare and Initialize

```
struct student
{
 char name[20];
 int roll;
 float marks;
} std1 = { "Pritesh",67,78.3 };
```

In the above code snippet, we have seen that structure is declared and as soon as after declaration we have initialized the structure variable.

```
std1 = { "Pritesh",67,78.3 }
```

This is the code for initializing structure variable in C programming

---

### 3. What is nested structure?

Ans :

Ans :

Structures within a structure means nesting of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
{
 char name;
 char department;
 int basic_pay;
 int dearness_allowance;
 int house_rent_allowance;
 int city_allowance;
}
employee;
```

This structure defines name, department, basic pay and three kinds of allowances. We can group all the items related to allowance together and declare them under a substructure as shown below :

```
struct salary
{
 char name;
 char department;
 struct
 {
 int dearness;
 int house_rent;
 }
 allowance ;
}
employee;
```

**4. Differentiate structures with arrays.***Ans :*

| Structures                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | Arrays                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>A structure is a collection of variables of heterogeneous data types.</p> <p>A structure is declared by using the keyword struct as follows :</p> <pre>struct structure_name {     type1field1;     type2field2;     :     typen fieldn; }</pre> <p>The individual members of the structure are accessed using the . (dot) operator.</p> <pre>Struct_variable.field_name</pre> <p><b>Rules to be followed while using structures</b></p> <ul style="list-style-type: none"> <li>Define structure</li> <li>Declare the structure variables to be used in program.</li> <li>Use '.' (dot) operator to access the members of the ordinary structure variable.</li> <li>Use * operator to access the members of the pointer structure variables.</li> </ul> | <p>An array is a collection of variables of homogeneous types.</p> <p>An array can declare without using any keyword but just by specifying the data type that must be followed by each element of the array and the maximum dimension of the array. This can be given as</p> <pre>datatype_arr_name[max_dimension];</pre> <p>The individual members of the array are accessed using the subscript. The number specified within the subscript specifies the element to be accessed. The maximum subscript value is max_dimension-1. The minimum subscript value is 0.</p> <p><b>Rules to be followed while using arrays</b></p> <ul style="list-style-type: none"> <li>Declare an array</li> <li>During declaration mention sizes of the dimensions</li> <li>To access the array elements use subscript.</li> <li>Do not access beyond the size of the array. It may give incorrect results.</li> <li>All the elements of an array must be of same data type.</li> </ul> |

**5. Define Union.***Ans :*

A union, is a collection of variables of different types, just like a structure. However, with unions, you can only store information in one field at any one time.

A union can also be viewed as a variable type that can contain many different variables, but only actually holds one of them at a time (not like a structure). This can save memory if you have a group of data where only one of the types is used at a time.

Unions like structure contain members whose individual data types may differ from one another. However the members that compose a union all share the same storage area within the computer's memory where as each member within a structure is assigned its own unique storage area. Thus unions are used to conserve memory. They are useful for application involving multiple members. Where values need not be assigned to all the members at any one time.



**Usage**

- access individual bytes of larger type
- variable format input records (coded records)
- sharing an area to save storage usage
- unions not used nearly as much as structures

Like structures union can be declared using the keyword union as follows:

```
union item
{
 int m;
 float p;
 char c;
}
code;
```

---

**6. Define enumeration. (or) Explain about enumerated data type.**

*Ans :*

An enumeration is a data type, similar to a structure or a union. Its members are constants that are written as identifiers, though they have signed integer values. These constants represent values that can be assigned to corresponding enumeration variables.

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the “enumeration set,” “enumerator constants,” “enumerators,” or “members”). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

Variables of **enum** type can be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the **#define** preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

**Syntax**

*enum-specifier :*

```
enum identifieropt { enumerator-list }
enum identifier
```

---

**7. List out the advantages of enumeration.**

*Ans :*

1. Enumeration constants are automatically assigned equivalent integer values, beginning with 0 for the first constant, with each successive constant increasing by 1, and it can be overridden to some other values.
2. Use of enumeration variables within a program can often increase the logical clarity of that program.
3. Enumeration variables are particularly used as flags to indicate various options for carrying out a calculation.

**8. What is the use of EOF?**

*Ans :*

End-of-file (commonly abbreviated EOF) is a condition in a computer operating system where no more data can be read from a data source. The data source is usually called a file or stream.

**Syntax**

```
int feof(FILE *stream);
```

EOF does –

- Macro tests if end-of-file has been reached on a stream.
- feof is a macro that tests the given stream for an end-of-file indicator.
- Once the indicator is set, read operations on the file return the indicator until rewind is called, or the file is closed.
- The end-of-file indicator is reset with each input operation.

**9. What is the use of fclose( ) function?**

*Ans :*

To close a file, use the fclose( ) function. The prototype of this function is -

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

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

**10. List the functions used for reading, writing text files.**

*Ans :*

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are simplest functions used to read and write individual characters to a file. C provides four functions that can be used to read text files from the disk. These are

- fscanf()
- fgets()
- fgetc()
- fread()

C provides four functions that can be used to write text files into the disk. These are

- fprintf()
- fputs()
- fputc()
- fwrite()

## Choose the Correct Answers

1. For accessing a structure element using a pointer, you must use? [ d ]  
(a) Pointer operator (&) (b) Dot operators(.)  
(c) Pointer operator(\*) (d) Arrow operator(->)
2. Which operator is used to connect structure name to its member name? [ a ]  
(a) dot operator(.) (b) logical operator(&&)  
(c) pointer operator(&) (d) Arrow operator(->)
3. In C, structure values can be passed as arguments to function by? [ d ]  
(a) passing each member of the structure as an argument of function call  
(b) passing copy of the entire structure to the called function  
(c) passing structure as an argument using pointer  
(d) All of above
4. What does the following segment of code do? [ a ]  
**fprintf(fp, "Copying!");**  
(a) It writes "Copying!" into the file pointed by fp  
(b) It reads "Copying!" from the file and prints on display  
(c) It writes as well as reads "Copying!" to and from the file and prints it  
(d) None of the mentioned
5. FILE reserved word is \_\_\_\_ [ d ]  
(a) A structure tag declared in stdio.h  
(b) One of the basic datatypes in c  
(c) Pointer to the structure defined in stdio.h  
(d) It is a type name defined in stdio.h
6. stdout, stdin and stderr are \_\_\_\_ [ a ]  
(a) File pointers (b) File descriptors  
(c) Streams (d) Structure
7. Which of the following reduces the size of a structure? [ b ]  
(a) union (b) bit-fields  
(c) malloc (d) None of the mentioned
8. For binary files, a \_\_\_\_ must be appended to the mode string. [ b ]  
(a) Nothing (b) "b"  
(c) "binary" (d) "01"

9. If there is any error while opening a file, fopen will return [ c ]  
(a) Nothing (b) EOF  
(c) NULL (d) Depends on compiler
10. Size of a union is determined by size of the \_\_\_\_\_. [ c ]  
(a) First member in the union  
(b) Last member in the union  
(c) Biggest member in the union  
(d) Sum of the sizes of all members
11. fputs function writes a string to a file that only ends with a newline. [ b ]  
(a) true (b) false  
(c) Depends on the standard (d) Depends on the compiler
12. What is the size of array "line" used in fgets(line, maxline, \*fp) function? [ b ]  
(a) maxline – 1 (b) maxline  
(c) maxline + 1 (d) Size is dynamic
13. Which of the following share a similarity in syntax? [ b ]  
1. Union, 2. Structure,  
3. Arrays and 4. Pointers  
(a) 3 and 4 (b) 1 and 2  
(c) 1 and 3 (d) 1, 3 and 4
14. In function free(p), p is a [ d ]  
(a) int (b) Pointer returned by malloc()  
(c) Pointer returned by calloc() (d) Both b & c

## Fill in the Blanks

1. The fscanf( ) statement reads data from \_\_\_\_\_ .
2. Command line arguments are used to accept argument from a \_\_\_\_\_ .
3. The redirection operator '>' transfers any output to \_\_\_\_\_ .
4. The EOF is equivalent to \_\_\_\_\_ .
5. \_\_\_\_\_ function is used to detect the end of file.
6. When fopen( ) fails to open a file it returns \_\_\_\_\_ .
7. The contents of a file will be lost if it is opened in \_\_\_\_\_ .
8. Size of void pointer is \_\_\_\_\_ .
9. Puts (argv[0]) ; prints \_\_\_\_\_ .
10. Size of a union is determined by size of the \_\_\_\_\_ .

### ANSWERS

1. file
2. command prompt of operating system
3. text file
4. non-zero
5. feof( )
6. -1
7. w and w+ mode
8. 2 byte.
9. the name of the executable code file.
10. biggest number in the union

## C - Programs

1. Write a program to find the largest two numbers using if and conditional operator.

Ans :

```
/* C Program to Find Largest of Two numbers */
#include <stdio.h>
int main() {
 int a, b;
 printf("Please Enter Two different values\n");
 scanf("%d %d", &a, &b);
 if(a > b)
 {
 printf("%d is Largest\n", a);
 }
 else if (b > a)
 {
 printf("%d is Largest\n", b);
 }
 else
 {
 printf("Both are Equal\n");
 }

 return 0;
}
```

Output :

```
Please Enter Two different values
34 23
34 is largest
```

2. Write a program to calculate arithmetic operations of two numbers using switch.

Ans :

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int a,b;
 int op;
 //clrscr();
 printf(" 1.Addition\n 2.Subtraction\n 3.Multiplication\n 4.Division\n");
```

```
printf("Enter the values of a & b: ");
scanf("%d %d",&a,&b);
printf("Enter your Choice : ");
scanf("%d",&op);
switch(op)
{
case 1 :
 printf("Sum of %d and %d is : %d",a,b,a+b);
 break;
case 2 :
 printf("Difference of %d and %d is : %d",a,b,a-b);
 break;
case 3 :
 printf("Multiplication of %d and %d is : %d",a,b,a*b);
 break;
case 4 :
 printf("Division of Two Numbers is %d : ",a/b);
 break;
 default :
 printf(" Enter Your Correct Choice.");
 break;
}
getch();
}
```

**Output :**

1. Addition
2. Subtraction
3. Multiplication
4. Division

Enter the values of a & b : 20    15

Enter your Choice : 1

Sum of 20 and 15 is : 35

**3. Write a C program to print the reverse of a given number.**

*Ans :*

```
#include <stdio.h>
int main()
{
 int n, reversedNumber = 0, remainder;
 printf("Enter an integer: ");
 scanf("%d", &n);
```

```
while(n != 0)
{
 remainder = n%10;
 reversedNumber = reversedNumber*10 + remainder;
 n /= 10;
}
printf("Reversed Number = %d", reversedNumber);
return 0;
}
```

**Output :**

Enter an interger : 123  
Reversed Number = 321

**4. Write a program to print whether the given number is a prime or not.**

*Ans :*

```
#include <stdio.h>
main() {
 int n, i, c = 0;
 printf("Enter any number n:");
 scanf("%d", &n);
 /*logic*/ for (i = 1; i <= n; i++) {
 if (n % i == 0) {
 c++;
 }
 }
 if (c == 2) {
 printf("n is a Prime number");
 }
 else {
 printf("n is not a Prime number");
 }
 return 0;
}
```

**Output :**

Enter any number n : 21  
n is not a Prime number  
Enter any number n : 17  
n is a Prime number

**5. Write a program to find largest and smallest elements in a given list of numbers**

*Ans :*

```
#include <stdio.h>
int main()
```



```

{
 int a[50],i,n,large,small;
 printf("How many elements:");
 scanf("%d",&n);
 printf("Enter the Array:");
 for(i=0;i<n; ++i)
 scanf("%d",&a[i]);
 large = small = a[0];
 for(i=1;i<n; ++i)
 {
 if(a[i]>large)
 large = a[i];
 if(a[i]<small)
 small = a[i];
 }
 printf("The largest element is %d",large);
 printf("\nThe smallest element is %d",small);
 return 0;
}

```

**Output**

How many elements:5  
Enter the Array:1 8 12 4 6  
The largest element is 12  
The smallest element is 1

**6. Write a program to find the sum of two matrices.***Ans :*

```

#include <stdio.h>
int main(){
 int r, c, a[100][100], b[100][100], sum[100][100], i, j;
 printf("Enter number of rows (between 1 and 100): ");
 scanf("%d", &r);
 printf("Enter number of columns (between 1 and 100): ");
 scanf("%d", &c);
 printf("\nEnter elements of 1st matrix:\n");
 for(i=0; i<r; ++i)
 for(j=0; j<c; ++j)
 {
 printf("Enter element a%d%d: ",i+1,j+1);

```

```
scanf("%d",&a[i][j]);
 }
printf("Enter elements of 2nd matrix:\n");
for(i=0; i<r; ++i)
for(j=0; j<c; ++j)
{
printf("Enter element a%d%d: ",i+1, j+1);
scanf("%d", &b[i][j]);
}
// Adding Two matrices
for(i=0;i<r; ++i)
for(j=0;j<c; ++j)
{
sum[i][j]=a[i][j]+b[i][j];
}
// Displaying the result
printf("\nSum of two matrix is: \n\n");
for(i=0;i<r; ++i)
for(j=0;j<c; ++j)
{
printf("%d ",sum[i][j]);
if(j==c-1)
{
printf("\n\n");
}
}
return 0;
}
```

**Output**

```
Enter number of rows (between 1 and 100) : 2
Enter number of columns (between 1 and 100) : 3
Enter elements of 1st matrix :
Enter element a11 : 2
Enter element a12 : 3
Enter element a13 : 4
Enter element a 21 : 5
Enter element a 22 : 2
Enter element a 23 : 3
Enter elements of 2nd matrix :
Enter element a 11 : - 4
Enter element a 12 : 5
Enter element a 13 : 3
```

Enter element a 21 : 5  
Enter element a 22 : 6  
Enter element a 23 : 3  
Sum of two matrix is :  
-2    8    7  
10    8    6

**7. Write a program to find the product of two matrices.**

**Ans :**

```
#include <stdio.h>
int main()
{
 int a[10][10], b[10][10], result[10][10], r1, c1, r2, c2, i, j, k;
 printf("Enter rows and column for first matrix: ");
 scanf("%d %d", &r1, &c1);
 printf("Enter rows and column for second matrix: ");
 scanf("%d %d", &r2, &c2);
 // Column of first matrix should be equal to column of second matrix and
 while (c1 != r2)
 {
 printf("Error! column of first matrix not equal to row of second.\n\n");
 printf("Enter rows and column for first matrix: ");
 scanf("%d %d", &r1, &c1);
 printf("Enter rows and column for second matrix: ");
 scanf("%d %d", &r2, &c2);
 }
 // Storing elements of first matrix.
 printf("\nEnter elements of matrix 1:\n");
 for(i=0; i<r1; ++i)
 for(j=0; j<c1; ++j)
 {
 printf("Enter elements a%d%d: ", i+1, j+1);
 scanf("%d", &a[i][j]);
 }
 // Storing elements of second matrix.
 printf("\nEnter elements of matrix 2:\n");
 for(i=0; i<r2; ++i)
 for(j=0; j<c2; ++j)
 {
 printf("Enter elements b%d%d: ", i+1, j+1);
 scanf("%d", &b[i][j]);
 }
```

```
 }
 // Initializing all elements of result matrix to 0
 for(i=0; i<r1; ++i)
 for(j=0; j<c2; ++j)
 {
 result[i][j] = 0;
 }
 // Multiplying matrices a and b and
 // storing result in result matrix
 for(i=0; i<r1; ++i)
 for(j=0; j<c2; ++j)
 for(k=0; k<c1; ++k)
 {
 result[i][j] += a[i][k]*b[k][j];
 }
 // Displaying the result
 printf("\nOutput Matrix:\n");
 for(i=0; i<r1; ++i)
 for(j=0; j<c2; ++j)
 {
 printf("%d ", result[i][j]);
 if(j == c2-1)
 printf("\n\n");
 }
 return 0;
}
```

**Output :**

```
Enter rows and column for first matrix : 3
2
Enter rows and column for second matrix : 3
2
Error! column of first matrix not equal to row of second.

Enter rows and column for first matrix : 2
3
Enter rows and column for second matrix : 3
2
Enter elements of matrix 1 :
Enter elements all : 3
Enter elements a 12 : -2
```

Enter elements a 13 : 5

Enter elements a 22 : 0

Enter elements a 23 : 4

Enter elements of matrix 2 :

Enter elements b 11 : 2

Enter elements b 12 : 3

Enter elements b 21 : -9

Enter elements b 22 : 0

Enter elements b 31 : 0

Enter elements b 32 : 4

Out Matrix :

24    29

6     25

**8. Write a program to print the reverse of a given string.**

*Ans :*

```
#include <stdio.h>
#include <string.h>
int main() {
 char str[100], temp;
 int i, j = 0;
 printf("\nEnter the string :");
 gets(str);
 i = 0;
 j = strlen(str) - 1;
 while (i < j) {
 temp = str[i];
 str[i] = str[j];
 str[j] = temp;
 i++;
 j--;
 }
 printf("\nReverse string is :%s", str);
 return (0);
}
```

**Output**

Enter the string : hello

Reverse string is : olleh

**9. Write a program to find the factorial of a positive integer using iteration and recursion.**

Ans :

```
#include <stdio.h>
// Iterative function to find factorial of a number using for loop
unsigned long factorial(int n)
{
 unsigned long fact = 1;
 int i;
 for (i = 1; i <= n; i++)
 fact = fact * i;
 return fact;
}
// Program to find factorial of a number
int main()
{
 int n = 5;
 printf("The Factorial of %d is %lu", n, factorial(n));
 return 0;
}
```

**Output :**

The factorial of 5 is 120

**10. Write a program to find the GCD of two positive integers using iteration and recursion.**

Ans :

```
* C program to find GCD (HCF) of two numbers using recursion
*/
#include <stdio.h>
/* Function declaration */
intgcd(int a, int b);
int main()
{
 int num1, num2, hcf;
 /* Input two numbers from user */
 printf("Enter any two numbers to find GCD: ");
 scanf("%d%d", &num1, &num2);
 hcf = gcd(num1, num2);
 printf("GCD of %d and %d = %d", num1, num2, hcf);
 return 0;
}
/**
* Recursive approach of euclidean algorithm to find GCD of two numbers
```

```
*/
intgcd(int a, int b)
{
 if(b == 0)
 return a;
 else
 returngcd(b, a%b);
}
```

**Output :**

Enter the two numbers to find their GCD : 100 70

The GCD of 100 and 70 is 10.

**11. Write a program to demonstrate the call by value and the call by reference concepts.**

Ans :

Call by value

```
#include <stdio.h>
#include <conio.h>
void swap(int a, int b)
{
 int temp;
 temp = a;
 a = b;
 b = temp;
}
void main()
{
 int a = 100, b = 200;
 clrscr();
 swap(a, b); // passing value to function
 printf("\nValue of a: %d", a);
 printf("\nValue of b: %d", b);
 getch();
}
```

Output

Value of a: 200

Value of b: 100

Call by reference

```
#include <stdio.h>
#include <conio.h>
void swap(int*a, int*b)
{
```

```
int temp;
temp = *a;
*a = *b;
*b = temp;
}
void main()
{
int a=100, b=200;
clrscr();
swap(&a,&b); // passing value to function
printf("\nValue of a: %d",a);
printf("\nValue of b: %d",b);
getch();
}
Output
Value of a: 200
Value of b: 100
```

**Out put :**

Value of a : 200  
Value of b : 100

**12. Write a program to illustrate the use of Enumeration data type.**

*Ans :*

```
// program to create enumerated data type for 7 days and display their values in integer constants.
#include <stdio.h>
int main()
{
enum week{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
printf("Sun = %d", Sun);
printf("\nMon = %d", Mon);
printf("\nTue = %d", Tue);
printf("\nWed = %d", Wed);
printf("\nThu = %d", Thu);
printf("\nFri = %d", Fri);
printf("\nSat = %d", Sat);
```



```
return 0;
```

```
}
```

Copy

Sample Output:

Sun = 0

Mon = 1

Tue = 2

Wed = 3

Thu = 4

Fri = 5

Sat = 6

**Output :**

Sun = 0

Mon = 1

Tue = 2

Web = 3

Thu = 4

Fri = 5

Sat = 6

**13. Write a program to illustrate the use of structure concept.**

*Ans :*

Example: Store Information in Structure and Display it

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

```
struct student
```

```
{
```

```
char name[50];
```

```
int roll;
```

```
float marks;
```

```
}s[10];
```

```
int main()
```

```
{
```

```
int i;
```

```
printf("Enter information of students:\n");
```

```
// storing information
```

```
for(i=0; i<10; ++i)
```

```
{
```

```
s[i].roll = i+1;
```

```
printf("\nFor roll number%d,\n",s[i].roll);
```

```
printf("Enter name: ");
scanf("%s",s[i].name);
printf("Enter marks: ");
scanf("%f",&s[i].marks);
printf("\n");
}
printf("Displaying Information:\n\n");
// displaying information
for(i=0; i<10; + +i)
{
printf("\nRoll number: %d\n",i+1);
printf("Name: ");
puts(s[i].name);
printf("Marks: %.1f",s[i].marks);
printf("\n");
}
return0;
}
```

**Output**

Enter information of students:

For roll number1,

Enter name: Tom

Enter marks: 98

For roll number2,

Enter name: Jerry

Enter marks: 89

.

.

.

Displaying Information:

Roll number: 1

Name: Tom

Marks: 98

.

**14. Write a program to illustrate the use of union concept.**

Ans :

```
/* Write a C program to illustrate the concept of unions*/
#include <stdio.h>
#include <conio.h>
void main()
{
 union number
 {
 int n1;
 float n2;
 };
 union number x;
 clrscr() ;
 printf("Enter the value of n1: ");
 scanf("%d", &x.n1);
 printf("Value of n1 = %d", x.n1);
 printf("\nEnter the value of n2: ");
 scanf("%d", &x.n2);
 printf("Value of n2 = %d\n",x.n2);
} /* End of main() */
/*_____
```

**Output**

```
Enter the value of n1: 2
Value of n1 = 2
Enter the value of n2: 3
Value of n2 = 0
```

**15. Write a program to write content into a file and display contents of a file.**

Ans :

```
#include <stdio.h>
int main()
{
 FILE *fp; /* file pointer*/
 char fName[20];
 printf("\nEnter file name to create :");
 scanf("%s",fName);
 /*creating (open) a file*/
 fp = fopen(fName,"w");
 /*check file created or not*/
 if(fp == NULL)
```

```

 {
 printf("File does not created!!!");
 exit(0); /*exit from program*/
 }
 printf("File created successfully.");
 /*writting into file*/
 putc('A',fp);
 putc('B',fp);
 putc('C',fp);
 printf("\nData written successfully.");
 fclose(fp);
 /*again open file to read data*/
 fp=fopen(fName,"r");
 if(fp==NULL)
 {
 printf("\nCan't open file!!!");
 exit(0);
 }
 printf("Contents of file is :\n");
 printf("%c",getc(fp));
 printf("%c",getc(fp));
 printf("%c",getc(fp));
 fclose(fp);
 return 0;
 }

```

**Output**

```

Enter file name to create : ok.txt
File created successfully.
Data written successfully.
Contents of file is :
ABC

```

**16. Write a program to copy content of one file into another file and display the content of new file.***Ans :*

```

#include <stdio.h>
#include <stdlib.h> // For exit()
intmain()
{
 FILE*fptr1, *fptr2;
 charfilename[100], c;
 printf("Enter the filename to open for reading \n");
 scanf("%s", filename);
 // Open one file for reading
 fptr1 = fopen(filename, "r");

```

```
 if(fp1 == NULL)
 {
 printf("Cannot open file %s \n", filename);
 exit(0);
 }
 printf("Enter the filename to open for writing \n");
 scanf("%s", filename);
 // Open another file for writing
 fp2 = fopen(filename, "w");
 if(fp2 == NULL)
 {
 printf("Cannot open file %s \n", filename);
 exit(0);
 }
 // Read contents from file
 c = fgetc(fp1);
 while(c != EOF)
 {
 fputc(c, fp2);
 c = fgetc(fp1);
 }
 printf("\nContents copied to %s", filename);
 fclose(fp1);
 fclose(fp2);
 return 0;
}
```

**Output:**

```
Enter the filename to open for reading
a.txt
Enter the filename to open for writing
b.txt
Contents copied to b.txt
```

FACULTY OF SCIENCE  
B.Sc. I - Semester (CBCS) Examination  
October / November - 2020  
PROGRAMMING IN C

Time : 2 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)**

**(Short Answer Type)**

**Note :** Answer any **FIVE** questions.

**ANSWERS**

1. Define CPU. Discuss its functions. (Unit-I, Q.No.17)
2. Explain flowchart with an example. (Unit-I, SQA.12, Q.No.28)
3. Define a variable. Write the rules that must be followed while creating a variable. (Unit-I, Q.No.47)
4. Write about escape sequences with its purpose. (Unit-II, Q.No.3)
5. What is the use of Break statement ? Demonstrate with a program. (Unit-II, Q.No.17(1<sup>st</sup> Point))
6. What is a string ? Write string functions with examples. (Unit-II, SQA.11)
7. What is a function ? Explain the process of function initialization. (Unit-III, Q.No.1)
8. Write about storage classes with its purpose. (Unit-III, Q.No.16)

**Ans :**

Storage Classes are used to describe about the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

9. Explain pointer to pointer concept with an example. (Unit-III, Q.No.31)
10. Differentiate structure and union.

**Ans :**

| S.No. | Structure                                                                                                                                                                                                                                    | S.No. | Union                                                                                                                                                                                  |
|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.    | The keyword struct is used to define a structure.                                                                                                                                                                                            | 1.    | The keyword union is used to define a union.                                                                                                                                           |
| 2.    | When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than (or) equal to the sum of sizes of its members. The smaller members may end with unused slack bytes. | 2.    | When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member. So, size of union is equal to the size of largest member. |
| 3.    | Each member within a structure is assigned unique storage area of location.                                                                                                                                                                  | 3.    | Memory allocated is shared by individual members of union.                                                                                                                             |

|    |                                                                                                                                         |    |                                                                                                                       |
|----|-----------------------------------------------------------------------------------------------------------------------------------------|----|-----------------------------------------------------------------------------------------------------------------------|
| 4. | The address of each member will be in ascending order This indicates that memory for each member will start at different offset values. | 4. | The address is same for all the members of a union. This indicates that every member begins at the same offset value. |
| 5. | Altering the value of a member will not affect other members of the structure.                                                          | 5. | Altering the value of any of the member will alter other member values.                                               |
| 6. | Individual member can be accessed at a time.                                                                                            | 6. | Only one member can be accessed at a time.                                                                            |
| 7. | Several members of a structure can initialize at once.                                                                                  | 7. | Only the first member of a union can be initialized.                                                                  |

11 . What is a Binary file ? Write its advantages and disadvantages.

(Unit-IV, Q.No.20)

Ans :

Advantage Keeping the binary data in the database and returning it in the record set is simpler. A disadvantage is that your database will now require more storage space and this will impact backups and other things. Sometimes having large amounts of binary data in a database can also have an impact on performance.

12. Write various file handling functions in C.

(Unit-IV, Q.No.19)

### PART - B (3 × 20 = 60 Marks)

#### (Essay Answer Type)

**Note :** Answer any **THREE** questions.

13. Discuss generations and categories of programming languages with examples. (Unit-I, Q.No.18,19)

14. (i) Describe structure of a C program. (Unit-I, Q.No.39)

(ii) Write a shortnotes on type conversions in C. (Unit-I, Q.No.53)

15. Explain IF-ELSE and nested IF-ELSE with illustrations and programs. (Unit-II, Q.No.7,8)

16. Define and represent multidimensional Array. Write a program to find the transpose of a matrix. (Unit-II, Q.No.21)

Ans :

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int a[10][10], m, n, i, j;
 clrscr();
 printf("Enter the order of matrix:");
 scanf("%d%d", &m, &n);
 printf("Enter elements of the matrix:");
```

```
for(i = 1; i <= m; i++)
{
 for(j = 1; j <= n; j++)
 {
 scanf("%d", &a[i][j]);
 }
}
printf("Transpose of the given matrix is:\n");
for(i = 1; i <= n; i++)
{
 for(j = 1; j <= m; j++)
 {
 printf("%d\t", a[j][i]);
 getch();
 }
 printf("\n");
}
```

- 
17. What is Recursion ? Explain with an example and program. (Unit-III, Q.No.19)
  18. Explain Direct Memory Access. Describe CALLOC ( ) and MALLOC ( ) function with a program. (Unit-III, Q.No.32)
  19. Define a structure with an example. Explain structure variable creation and accessing of structure elements. (Unit-IV, Q.No.1,2,4,)
  20. What is the use of File ? Write a C program to create, write and read data from the file. (Unit-IV, Q.No.16,19)



**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination****October / November - 2019****PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (8 × 4 = 32 Marks)****(Short Answer Type)****Note :** Answer any **EIGHT** of the following questions.**ANSWERS**

1. Define computer. Explain its parts. (Unit-I, SQA.1,3)
2. What is an algorithm ? Write an algorithm to find greatest of two numbers. (Unit-I, Q.No.24)

**Ans :**

- Step 1 : Start
- Step 2 : Read two numbers X and Y.
- Step 3 : Find the greater number out of X and Y. and store the result as 'Z'
- Step 4 : Display 'Z'
- Step 5 : Stop

3. Describe expression evaluation precedence and associativity with an example. (Unit-I, Q.No.52)
4. Write about escape sequences with its purpose. (Unit-II, SQA.2)
5. Write a program to demonstrate conditional operator. (Unit-II, SQA.4)
6. Explain the functions from ctype.h.

**Ans :**

It is a header file in standard library which is used to handle character type such as digital, alphanumeric, control characters etc.

**Syntax**

#include &lt;ctype.h&gt;

ctype.h header file contains the below given functions.

- (i) **toupper()** : It is a function used to convert the lowercase characters to uppercase.

**Syntax :** datatype toupper();

- (ii) **tolower()** : It is used to convert the uppercase characters to lowercase.

**Syntax :** datatype tolower();

- (iii) **isalnum()** : It is used to determine whether the argument is letter or digit.

**Syntax :** datatype isalnum();

- (iv) **isalpha()** : It checks whether the given argument is letter or not.

**Syntax :** datatype isalpha();

7. Write a program to implement CALL BY REFERENCE. (Unit-III, SQA.5)
8. What is Inline function in C ? Write the advantages of Inline functions. (Unit-III, Q.No.18)
9. Define a pointer. Write the advantages of pointers. (Unit-III, SQA.10)
10. What is a structure ? How to create a structure? (Unit-IV, SQA.1)
11. Explain various operations on files. (Unit-IV, Q.No.17)
12. What is a Binary file ? Write its advantages. (Unit-IV, Q.No.20)

**PART - B (4 × 12 = 48 Marks)****(Essay Answer Type)**

**Note :** Answer **ALL** the following questions.

13. (a) Discuss:
  - (i) Types of computers (Unit-I, Q.No.5)
  - (ii) C variables and constants. (Unit-I, Q.No.47, 48)
- OR
- (b) Describe different data types with examples and programs for each. (Unit-I, Q.No.46)
14. (a) Explain all iterative statements with programs and illustrations for each. (Unit-II, Q.No.14,15,16)
- OR
- (b) Define and represent multidimensional array. Write a program to add (Unit-II, Q.No.21)
15. (a) Explain different storage classes with purpose, scope and program for each. (Unit-III, Q.No.16)
- OR
- (b) Explain Direct Memory Access. Describe CALLOC ( ) and MALLOC ( ) functions with a program. (Unit-III, Q.No.32)
16. (a) What is a Union ? Explain Union declaration and initialization and accessing with examples and programs. (Unit-IV, Q.No.9)

**Ans :**

Declaration of union variables and union is similar to that of structure variables and structures respectively.

Union can also be declared as shown below,

```
union <tag>
{
 datatype member 1;
 datatype member2;
 :
 datatype member n;
```

```
}
```

```
variable 1, variable 2,..., variablen;
```

The tag is optional in this declaration.

- Here, union is a keyword.
- member 1, member 2,..., member n are members of a union.
- The variables of union are declared as union variable 1, variable 2,..., variablen.

**Example**

```
union item
```

```
{
```

```
int a;
```

```
float b;
```

```
char ch;
```

```
}u;
```

In the above example,

- item is the name of the union.
- a, b and ch are members of the union.
- u is the union variable.

**Initialization**

A union member is initialized by assigning a value to it. The type of value should be same as the type of variable.

**Syntax**

```
variable.member = constant;
```

In C, only first member of the union can be initialized while declaring it.

If the union member is of integer type the value assigned to it should also be an integer. Otherwise the initialization is not valid. Union provides a convenience to the programmer to name a cell to store a type independent value. It might be either done by certain functions or by a macro containing various types.

**Example**

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

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

```
union un
```

```
{
```

```
int a;
```

```
}x={5};
```

```
int main()
```

```
{
```

```
clrscr();
```

```
printf("\n Value of a: %d\\ x.a);
```

```
getch();
```

```
return 0;)
```

**Accessing**

Accessing of members of a union is done similar to that of a structure. It is done by using dot(.) operator for unions and using operator for pointer to unions.

**Syntax**

Union members can be accessed in the following way,  
variable 1 .member 1 (or)  
variable 1 → member 1 (In case of pointer to a union).

**Example**

Consider the above example,  
student 2.age = 20  
Here, student2 is the union variable and the age is the union member.  
Union members can also be accessed as,  
student 2 → age

**Program**

```
#include <stdio.h>
#include <string.h> union un
{
 int a;
 float b;
 char str[10];
};
int main()
{
 union un u;
 clrscr();
 u.a = 10;
 printf("u.a : %d\n", u.a);
 u.b = 65.2;
 printf("u.b : %f\n", u.b);
 strcpy(u.str, "C Programming");
 printf("u.str : %s\n", u.str);
 getch(); return 0;
```

OR

- (b) What is a text file ? Write a C program to create, write and read data from the file.

(Unit-IV, Q.No.19)

**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination**

May / June - 2019

**PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

1. Write about generations of programming languages. (Unit-I, Q.No.19)
2. Define the terms,
  - (a) Compiler (Unit-I, SQA.9)
  - (b) Interpreter (Unit-I, Q.No.20)
  - (c) Assembler (Unit-I, Q.No.20)
  - (d) Translator

**Ans :**

A translator is a programming language processor that converts a computer program from one language to another. It takes a program written in source code and converts it into machine code. It discovers and identifies the error during translation.

**Purpose**

It translates high-level language program into a machine language program that the central processing unit (CPU) can understand. It also detects errors in the program.

3. List out the escape sequences.
4. Define Array and give the syntax of declaration of 1D and 2D Arrays. (Unit-II, Q.No.18,19,21)
5. Explain an Inline functions with example. (Unit-III, Q.No.18)
6. Explain in detail Recursion. (Unit-III, Q.No.19)
7. Write the syntax of structure declaration. (Unit-IV, Q.No.2)
8. What are enumerated data types ? (Unit-IV, SQA.6)

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Answer **ALL** questions.

9. (a) Explain Generations and classifications of programming Language in detail.

(Unit-I, Q.No.18)

OR

- (b) Write the structure of C program and explain the expression evaluation, precedence and associativity in detail. (Unit-I, Q.No.39,51,52)

10. (a) Explain about formatted and non-formatted Input/Output statements in detail. (Unit-II, Q.No.2)

OR

- (b) (i) Define string and explain the syntax of declaration of character array, (Unit-II, Q.No.23,24)  
(ii) Explain in detail about iterative statements. (Unit-II, Q.No.14,15,16)

11. (a) Differentiate between call by value and call by reference with examples. (Unit-III, Q.No.12)

OR

- (b) Define pointer and explain about Arrays of pointers. (Unit-III, Q.No.22,29)

12. (a) Explain in detail about user-defined data types. (Unit-IV, Q.No.1,2)

OR

- (b) Explain about files in 'C' and working of binary file in 'C' language. (Unit-IV, Q.No.12,20)

**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination**

November / December - 2018

**PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

- |                                                         |                       |
|---------------------------------------------------------|-----------------------|
| 1. Explain briefly about memory hierarchy.              | (Unit-I, Q.No.11)     |
| 2. Explain the structure of 'C' program.                | (Unit-I, Q.No.39)     |
| 3. Write in detail about switch statement.              | (Unit-II, Q.No.10)    |
| 4. What are strings? Name some function in string.h.    | (Unit-II, Q.No.23,27) |
| 5. Write about storage classes in detail.               | (Unit-III, Q.No.16)   |
| 6. Define function and explain the concept of function. | (Unit-III, Q.No.1)    |
| 7. Define union and give the syntax of union.           | (Unit-IV, SQA.5)      |
| 8. Write some file management functions.                | (Unit-IV, Q.No.22)    |

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Attempt **ALL** questions.

- |                                                                                             |                          |
|---------------------------------------------------------------------------------------------|--------------------------|
| 9. (a) Define algorithm and explain in detail about different ways of stating an algorithm. | (Unit-I, Q.No.24,25,29)  |
| OR                                                                                          |                          |
| (b) Write about 'C' tokens and operators of 'C' language in detail.                         | (Unit-I, Q.No.44,50)     |
| 10. (a) Explain in detail about selection statements.                                       | (Unit-II, Q.No.5)        |
| (b) Explain the formatted input/output statements.                                          | (Unit-II, Q.No.2)        |
| OR                                                                                          |                          |
| (c) Define array. Explain in detail about 1-D and 2-D array declaration with example.       | (Unit-II, Q.No.18,19,21) |
| 11. (a) Define function. Explain in detail about passing arrays to function with example.   | (Unit-III, Q.No.1,13)    |
| OR                                                                                          |                          |
| (b) Explain in detail the pointers and its uses.                                            | (Unit-III, Q.No.22,28)   |

12. (a) Explain about structures and its members in detail. (Unit-IV, Q.No.2)

(b) What are enumerated data types ? (Unit-IV, Q.No.11)

OR

(c) Explain in detail about working of files in 'C' language. (Unit-IV, Q.No.19)



**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination**

May / June - 2018

**PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

1. Give the classification of computers. (Unit-I, Q.No.5)
2. Define, (i) Compiler (ii) Interpreter (Unit-I, Q.No.20)
3. Write about special control statement 'break' with example. (Unit-II, Q.No.17(Point-1))
4. Write about non-formatted I/O functions. (Unit-II, Q.No.2(Point-1))
5. What is a function? What are the uses of functions? (Unit-III, SQA.1,2)
6. What is Recursion? Explain with a program. (Unit-III, SQA.9)
7. Write about Enumerated data types. (Unit-IV, SQA.6)
8. Explain about accessing of records in files. (Unit-IV, Q.No.21)

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Attempt **ALL** the questions.

9. (a) (i) What is flowchart? Write about symbols used in flowchart with example. (Unit-I, Q.No.26)
- (ii) Write in detail about, (Unit-I, Q.No.50,51,52)
- (a) Operators and expressions (b) Precedence and associativity.
- OR
- (b) (i) Explain the Anatomy of a computer. (Unit-I, Q.No.6)
- (ii) Write in detail about developing program and software development process. (Unit-I, Q.No.21,22)
10. (a) Explain about formatted I/O functions and escape sequences. (Unit-II, Q.No.2(Point-2), Q.No.3)
- OR
- (b) Write about, (Unit-II, Q.No.14,15,16)
- (i) while statement (ii) do-while (iii) for statements with examples
11. (a) (i) Explain about scope of variables using example. (Unit-III, Q.No.14)

- (ii) Explain about pointers and strings with example. (Unit-III, Q.No.30)

OR

- (b) (i) What are storage classes ? Explain with examples. (Unit-III, Q.No.17)  
(ii) Explain about pointers to pointers, array of pointers. (Unit-III, Q.No.29,31)
12. (a) Differentiate between structure and union with example. (Unit-IV, Q.No.10)

OR

- (b) Write in detail about File Management function in 'C' language. (Unit-IV, Q.No.22)

**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination****November / December - 2017****PROGRAMMING IN C****Time : 3 Hours]****[Max. Marks :80****PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

- |                                                                           |                                           |
|---------------------------------------------------------------------------|-------------------------------------------|
| 1. Explain the operations of CPU.                                         | (Unit-I, Q.No.17)                         |
| 2. Give the classification of programming Languages.                      | (Unit-I, Q.No.19)                         |
| 3. Write about 'continue' statement with example.                         | (Unit-II, Q.No.17(2 <sup>nd</sup> Point)) |
| 4. Define string. Write the built-in functions in 'string.h' header file. | (Unit-II, SQA.11,27)                      |
| 5. Define pointer. What is the use of &, * in the pointer concept.        | (Unit-III, Q.No.22)                       |
| 6. What is Inline function? Explain with a program.                       | (Unit-III, Q.No.18)                       |
| 7. Briefly explain about union.                                           | (Unit-IV, SQA.5)                          |
| 8. What is binary file in 'C' ?                                           | (Unit-IV, Q.No.20)                        |

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Attempt **ALL** the questions.

- |                                                                                 |                            |
|---------------------------------------------------------------------------------|----------------------------|
| 9. (a) Write in detail about structure of 'C' program.                          | (Unit-I, Q.No.39)          |
| (b) Define Algorithm. Explain about step-form and pseudo-code.                  | (Unit-I, Q.No.24,29)       |
| OR                                                                              |                            |
| (c) Explain about memory hierarchy.                                             | (Unit-I, Q.No.11,12,13,14) |
| (d) Write in detail about,                                                      |                            |
| (i) C tokens                                                                    | (Unit-I, Q.No.44)          |
| (ii) Identifiers                                                                | (Unit-I, Q.No.45)          |
| (iii) Data types                                                                | (Unit-I, Q.No.46)          |
| (iv) Variables and constants.                                                   | (Unit-I, Q.No.47,48)       |
| 10. (a) Explain about special control statements with syntax and examples.      | (Unit-II, Q.No.17)         |
| OR                                                                              |                            |
| (b) Explain about 2D arrays. Write a program to multiplication of two matrices. | (Unit-II, Q.No.21)         |

11. (a) Give a short note on Dynamic memory allocation. (Unit-III, Q.No.32)  
(b) Explain about “passing array to functions” with example. (Unit-III, Q.No.13)  
OR  
(c) Explain about, ‘Call by value’ and ‘Call by reference’ with examples. (Unit-III, Q.No.11)
12. (a) Write in detail about files in ‘C’ language. (Unit-IV, Q.No.12)  
OR  
(b) Explain structure in ‘C’ language with example. (Unit-IV, Q.No.1)

**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination**

May / June - 2017

**PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

1. Define an operating system. Explain common functions of an operating system. (Unit-I, Q.No.15)
2. Explain the general structure of a C program. (Unit-I, Q.No.39)
3. Write a C program to find greatest of two numbers using if-else statement. (Unit-II, Q.No.20)
4. Explain character functions in <ctype.h> header file. (Refer - Nov/Dec.-2019,Q.No.6)
5. Write short notes on global and local variables. (Unit-III, Q.No.14)
6. What is memory leak?

**Ans :**

Memory leak occurs when programmers create a memory in heap and forget to delete it. Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

7. Write short note on enumeration data types. (Unit-IV, SQA.6)
8. Explain functions to delete and rename a file. (Unit-IV, Q.No.22)

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Attempt **ALL** the questions.

9. (a) Define a software. Explain system software and application software. (Unit-I, Q.No.10)
  - (b) Explain generation of programming languages. (Unit-I, Q.No.19)
- OR
- (c) Define flowchart. Explain guidelines, advantages and limitations of using flowcharts. (Unit-I, Q.No.26,27)
  - (d) Explain data types in C. (Unit-I, Q.No.46)
10. (a) Explain if-else if statement and switch statement with syntax and flowchart. (Unit-II, Q.No.9,10)
  - (b) Explain for loop with syntax and example program. (Unit-II, Q.No.16)

OR

- (c) Define array. Explain types of arrays with example. (Unit-II, Q.No.18,19,21)
- (d) Write a program to demonstrate addition and subtraction of a matrix. (Unit-II, Q.No.21)
11. (a) Define function. Explain the process of using functions. (Unit-III, Q.No.4)
- (b) Explain call by value with an example program. (Unit-III, Q.No.11)
- OR
- (c) What is a pointer ? What are the uses of pointers in C ? (Unit-III, Q.No.22,28)
- (d) Explain dynamic memory allocation and its functions. (Unit-III, Q.No.32)
12. (a) Explain array of structures with an example program. (Unit-IV, Q.No.5)
- (b) Explain declaring, initializing and accessing the members of a union. (Unit-IV, Q.No.2)
- OR
- (c) Explain four essential actions required to use a file. (Unit-IV, Q.No.18)
- (d) Explain functions used for random access to files of records. (Unit-IV, Q.No.21)

**FACULTY OF SCIENCE****B.Sc. I - Semester (CBCS) Examination**

December - 2016

**PROGRAMMING IN C**

Time : 3 Hours]

[Max. Marks :80

**PART - A (5 × 4 = 20 Marks)****(Short Answer Type)****Note :** Answer any **FIVE** of the following questions.**ANSWERS**

- |                                                                  |                    |
|------------------------------------------------------------------|--------------------|
| 1. What is a computer ? Explain the classification of computers. | (Unit-I, Q.No.1,5) |
| 2. Explain the different types of program statements in C.       | (Unit-I, Q.No.42)  |
| 3. Explain conditional operator in C with an example program.    | (Unit-II Q.No.12)  |
| 4. What is an array ? Explain with an example.                   | (Unit-II, SQA.8)   |
| 5. Write short notes on functions.                               | (Unit-III, SQA.1)  |
| 6. Write short notes on malloc and calloc functions.             | (Unit-III, SQA.14) |
| 7. Differentiate between structure and unions.                   | (Unit-IV, Q.No.10) |
| 8. Explain stream I/O model in files.                            | (Unit-IV, Q.No.19) |

**PART - B (4 × 15 = 60 Marks)****(Essay Answer Type)****Note :** Attempt **ALL** the questions.

- |                                                                                                                   |                          |
|-------------------------------------------------------------------------------------------------------------------|--------------------------|
| 9. (a) Explain major hardware components of a computer in detail.                                                 | (Unit-I, Q.No.1)         |
| OR                                                                                                                |                          |
| (b) Explain types of operators in C with example programs.                                                        | (Unit-I, Q.No.50)        |
| 10. (a) Explain for loop, while loop, do-while loop in C with an example program each.                            | (Unit-II, Q.No.14,15,16) |
| OR                                                                                                                |                          |
| (b) Explain string function in C with example program.                                                            | (Unit-II, Q.No.27)       |
| 11. (a) Explain storage classes in C with an example program each.                                                | (Unit-III, Q.No.16)      |
| OR                                                                                                                |                          |
| (b) Explain declaration, initialization and accessing of pointers in C with an example program.                   | (Unit-III, Q.No.23,24)   |
| 12. (a) Explain declaration, initialization and accessing of structure data type with syntax and example program. | (Unit-IV, Q.No.2,3,4)    |
| OR                                                                                                                |                          |
| (b) Explain functions used to work with text files for reading and writing text.                                  | (Unit-IV, Q.No.19)       |

**FACULTY OF SCIENCE**  
**B.Sc. I-Year I-Semester (CBCS) Examination**  
**COMPUTER SCIENCE**  
**MODEL PAPER - I**  
**Paper - I: Programming in C**

Time: 3 Hours

Max. Marks: 80

**PART – A (5 × 4 = 20 Marks)**

**[Short Answer Type]**

**Note: Answer any FIVE of the following questions**

1. List the Benefits of HLL. (Unit - I, SQA.8)
2. Define Operating System and its types. (Unit - I, SQA.6)
3. What is the use of goto statement in C ? (Unit - II, SQA.7)
4. What is the use of strcpy()? (Unit - II, SQA.13)
5. What is the use of address operator? (Unit - III, SQA.11)
6. Difference between call-by-value and call by reference. (Unit - III, SQA.6)
7. Define Union. (Unit - IV, SQA.5)
8. What is nested structure? (Unit - IV, SQA.3)
9. Explain about RAM. (Unit - I, SQA.5)
10. Explain if-else statement. (Unit - II, SQA.3)
11. Explain function with argument and return value. (Unit - III, SQA.3)
12. What is the use of EOF ? (Unit - IV, SQA.8)

**PART – B (4 × 15 = 60 Marks)**

**[Essay Answer Type]**

**Note: Attempt ALL the questions**

13. (a) (i) Write in detail about structure of 'C' program. (Unit - I, Q.No. 39)  
(ii) Define Algorithm. Explain about step-form and pseudo-code. (Unit - I, Q.No. 25)
- OR
- (b) (i) Explain about memory hierarchy. (Unit - I, Q.No. 11,13,14)  
(ii) Write in detail about : (Unit - I, Q.No. 44,45,46,47,48)
  - (a) C tokens
  - (b) Identifiers
  - (c) Data types
  - (d) Variables and constants



14. (a) Explain about control statements with syntax and examples. (Unit - II, Q.No. 4, To 17)

OR

- (b) Explain about 2D arrays? Write a program to multiplication of two matrices. (Unit - II, Q.No. 21, Lab Program - 7)

15. (a) (i) Give a short note on Dynamic memory allocation. (Unit - III, Q.No. 32)

- (ii) Explain about "passing Array to functions" with example. (Unit - III, Q.No. 29)

OR

- (b) Explain about, 'Call by value' and 'Call by reference' with examples. (Unit - III, Q.No. 11)

16. (a) Write in detail about files in 'C' language. (Unit - IV, Q.No. 12)

OR

- (b) Explain structures in 'C' language with example. (Unit - IV, Q.No. 1,2,3,4)

**FACULTY OF SCIENCE**  
**B.Sc. I-Year I-Semester (CBCS) Examination**  
**COMPUTER SCIENCE**  
**MODEL PAPER - II**  
**Paper - I: Programming in C**

Time: 3 Hours

Max. Marks: 80

**PART – A (5 × 4 = 20 Marks)****[Short Answer Type]****Note: Answer any FIVE of the following questions**

1. What are the differences between Compiler & Interpreter (Unit - I, SQA.10)
2. Explain different components of computers. (Unit - I, SQA.3)
3. Explain Conditional Operator. (Unit - II, SQA.4)
4. What are the uses of Dynamic Array? (Unit - II, SQA.10)
5. Define function. List out the advantages of functions. (Unit - III, SQA.1,2)
6. Explain about recursive function. (Unit - III, SQA.9)
7. Differentiate structures with arrays. (Unit - IV, SQA.6)
8. List the functions used for reading, writing text files. (Unit - IV, SQA.10)
9. What is the use of Compiler ? (Unit - I, SQA.9)
10. Explain do-while loop. (Unit - II, SQA.5)
11. Difference between call-by-value and call by reference. (Unit - III, SQA.6)
12. Differentiate structures with arrays. (Unit - IV, SQA.4)

**PART – B (4 × 15 = 60 Marks)****[Essay Answer Type]****Note: Attempt ALL the questions**

13. (a) (i) What is flow chart? Write about symbols used in flow chart with example. (Unit - I, Q.No. 26)  
(ii) Write in detail about. (Unit - I, Q.No. 50,51,52)  
(a) Operators and expressions  
(b) Precedence and associability.  
OR  
(b) (i) Explain the Anatomy of a computer. (Unit - I, Q.No. 6)  
(ii) Write in detail about developing program and software development process. (Unit - I, Q.No. 36)
14. (a) Explain about formatted I/O functions and escape sequences. (Unit - II, Q.No. 2,3)

OR

- (b) Write about: (Unit - II, Q.No. 13,14,15,16)
- (i) while statement
  - (ii) do-while
  - (iii) for statements with examples.
15. (a) (i) Explain about scope of variables using example. (Unit - III, Q.No. 14,15)
- (ii) Explain about pointers and strings with example. (Unit - III, Q.No. 30)
- OR
- (b) (i) What are storage classes ? Explain with examples. (Unit - III, Q.No. 17)
- (ii) Explain about pointers to pointers, Array of pointers. (Unit - III, Q.No. 29,31)
16. (a) Differentiate between structure and union with example. (Unit - IV, Q.No. 10)
- OR
- (b) Write in detail about File Management function in 'C' language. (Unit - IV, Q.No. 22)

**FACULTY OF SCIENCE**  
**B.Sc. I-Year I-Semester (CBCS) Examination**  
**COMPUTER SCIENCE**  
**MODEL PAPER - III**  
**Paper - I: Programming in C**

Time: 3 Hours

Max. Marks: 80

**PART – A (5 × 4 = 20 Marks)****[Short Answer Type]****Note: Answer any FIVE of the following questions**

1. Define Computer. List out the capability of computer. (Unit - I, SQA. 1,2)
2. What is cache memory? Explain its advantages & disadvantages. (Unit - I, SQA. 4)
3. Explain escape sequences in C - languages? (Unit - II, SQA. 2)
4. Define string and explain string declaration. (Unit - II, SQA. 11)
5. Differentiate malloc and calloc methods. (Unit - III, SQA. 14)
6. Define pointer. and list out the its benefits. (Unit - III, SQA. 10)
7. Define Structure. (Unit - IV, SQA. 1)
8. What is the use of EOF? What is the use of fclose() function? (Unit - IV, SQA. 8,9)
9. Define ALU and CU. (Unit - I, SQA.7)
10. How can we combine two strings. (Unit - II, SQA.12)
11. Explain about call by value method to pass arguments. (Unit - III, SQA.4)
12. What are Global Variables ? (Unit - IV, SQA.7)

**PART – B (4 x 15 = 60 Marks)****[Essay Answer Type]****Note: Attempt ALL the questions**

13. (a) (i) List the characteristics of computer ? Explain. (Unit - I, Q.No. 2)  
(ii) Explain different types of operating systems ? (Unit - I, Q.No. 16 )  
OR  
(b) (i) What do you mean by datatype ? How many data types does C - Language consist ? (Unit - I, Q.No. 46)  
(ii) What are constants ? Explain its types? (Unit - I, Q.No. 48)
14. (a) (i) Explain non-formatted & formatted I/O operations ? (Unit - II, Q.No. 2)  
OR  
(ii) Explain Special statement in loops of C - language. (Unit - II, Q.No. 17)

- (b) (i) Explain the declaration and Accessing of one dimensional array ? (Unit - II, Q.No. 19)
- (ii) What are string handling functions ? Explain with examples ? (Unit - II, Q.No. 27)
15. (a) (i) What are functions ? Explain the elements of functions.  
Explain the implementation of functions in C - language ? (Unit - III, Q.No. 1,3,4)
- (ii) Explain about arguments with no return values ? Explain arguments with return values ? (Unit - III, Q.No. 8,9)

**OR**

- (b) What are pointers ? Explain in detail with examples ? How pointers initialized ? (Unit - III, Q.No. 22,24)
16. (a) (i) Explain the process of Accessing Members of a Structure. (Unit - IV, Q.No. 4)
- (ii) Explain array of structures with example ? (Unit - IV, Q.No. 5)

**OR**

- (b) Explain the functions used in reading and writing of text files ? (Unit - IV, Q.No. 19)