MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

PROGRAMMING FOR PROBLEM SOLVING

DIGITAL NOTES

B.Tech – I Year – I Semester

R-20

Preface

C is a general-purpose programming language that is extremely popular, simple and flexible. It is machine-independent, structured programming language which is used extensively in various applications. C was the basic language to write everything from operating systems (Windows and many others) to complex programs like the Oracle database, Python interpreter and more.

It is said that 'C' is a god's programming language. One can say, C is a base for the programming. If you know 'C,' you can easily grasp the knowledge of the other programming languages that uses the concept of 'C'

'C' is a structured programming language in which program is divided into various modules. Each module can be written separately and together it forms a single 'C' program. This structure makes it easy for testing, maintaining and debugging processes.

This digital notes is designed to be a stand-alone introduction to C, even if you've never programmed before. This notes has been organized to meet syllabi requirements of both JNTUH and AICTE. It serves as an introductory material to first year students, enabling them to understand basic concepts of C and updating them on the problem solving ability.

Organization of the book

The content has been divided into five units, as per the syllabus.

Unit I – Introduction to Computer System, Problem Solving with algorithms and flowcharts
Basics of C Language – History, Structure, Tokens, Data types, Control structures

Unit II - Pointer basics, Arrays and Strings

Unit III – Modular Programming using functions, Dynamic Memory Management

Unit IV – Structures, Unions and Files

Unit V – Basic Data Structures – Stacks, Queues and Linked Lists

We would like to extend our sincere gratitude to Dr. VSK Reddy, Principal, Malla Reddy College of Engineering and Technology (autonomous) under whose patronage we were able to write this book. We are also indebted to Dr.D.Sujatha, Head of the Department, Computer Science and Engineering for her constant support and motivation for our academic growth. With great pleasure we acknowledge the compatible environment shared by our colleagues

I Year B. Tech CSE -I Sem

L	T/P/D	C
4	-/-/-	3

PROGRAMMING FOR PROBLEM SOLVING USING C

COURSE OBJECTIVES:

The students will be able to

1. Understand the use of computer system in problem solving and to build program logic with algorithms and flowcharts.
2. Explain the features and constructs of C programming such as data types, expressionsLoops, arrays, strings and pointers
3. Learn how to write modular Programs using Functions
4. Understand the use of Structures, Unions and Files
5. Use basic data structures like stacks, queues and linked lists in designing applications

UNIT - I

Introduction to Computing – Computer Systems, Computing Environments, Computer Languages, Algorithms and Flowcharts, Steps for Creating and Running programs.

Introduction to C – History of C, Features of C, Structure of C Program, Character Set, C Tokens-keywords, Identifiers, Constants, Data types, Variables. Operators, Expressions, Precedence and Associativity, Expression Evaluation, Type conversion, typedef, enum

Control Structures: Selection Statements (Decision Making) – if and switch statements, Repetition Statements (Loops) - while, for, do-while statements, Unconditional Statements – break, continue, goto. Command line arguments.

UNIT-II

Pointers – Pointer variable, pointer declaration, Initialization of pointer, accessing variables through pointers, pointer arithmetic, pointers to pointers, void pointers

Arrays – Definition, declaration of array, Initialization, storing values in array, two dimensional arrays, Multi-dimensional arrays.Arrays and Pointers, Array of pointers

Strings – Declaration and Initialization, String Input / Output functions, Array of strings, String manipulation functions, Unformatted I/O functions, strings and pointers

UNIT-III

Designing Structured Programs using Functions - Types of Functions- user defined functions, Standard Functions, Categories of functions, Parameter Passing techniques, Scope – Local Vs Global, Storage classes, Recursive functions.

Passing arrays as parameters to functions, Pointers to functions, Dynamic Memory allocation**,**

UNIT-IV

Structures and Unions - Declaration, initialization, accessing structures, operations on structures, structures containing arrays, structures containing pointers, nested structures, self referential structures, array of structures, structures and functions, structures and pointers, unions.

Files – Concept of a file, Streams, Text files and Binary files, Opening and Closing files,

File input / output functions.Sequential Access and Random Access Functions

UNIT-V

Basic Data Structures – Linear and Non Linear Structures – Implementation of Stacks, Queues, Linked Lists and their applications.

Case Studies

Case 1: Student Record Management System

The main features of this project include basic file handling operations; you will learn how to add, list, modify and delete data to/from file.

Currently, listed below are the only features that make up this project, but you can add new features as you like to make this project a better one!

- Add record
- List record
- Modify record
- Delete record

Case 2: Library Management System

This project has 2 modules.

1. Section for a librarian
2. Section for a student

A librarian can add, search, edit and delete books. This section is password protected. That means you need administrative credentials to log in as a librarian.

A student can search for the book and check the status of the book if it is available.

Here is list of features that you can add to the project.

1. You can create a structure fora student that uniquely identify eachstudent. When a student borrows a book from the library, you link his ID to Book ID so that librarian can find how burrowed particular book.
2. You can create a feature to bulk import the books from CSV file.
3. You can add REGEX to search so that a book can be searched using ID, title, author or any of the field.
4. You can add the student login section.

TEXT BOOKS:

1. Mastering C, K.R.Venugopal, S R Prasad, Tata McGraw-Hill Education.
2. Computer Science: A Structured Programming Approach Using C, B.A.Forouzan and R.F. Gilberg, Third Edition, Cengage Learning
3. Data Structures and Algorithms Made Easy by NarasimhaKarumanchi, Career Monk publications, 2017

REFERENCE BOOKS:

1. The C Programming Language, B.W. Kernighan and Dennis M.Ritchie, PHI.
2. Computer Programming, E.Balagurusamy, First Edition, TMH.
3. C and Data structures – P. Padmanabham, Third Edition, B.S. Publications.
4. Programming in *C, Ashok Kamthane*. Pearson Education India.
5. Let us C ,YashwanthKanethkar, 13th Edition, BPB Publications.
6. Data Structures using C by Aaron M. Tenenbaum, Pearson Publications
7. Data Structures using C by Puntambekar

COURSE OUTCOMES:

At the end of the course the student will be able to

1. Understand a problem and build an algorithm/flowchart to solve it
2. Define variables and construct expressions using C language
3. Construct C programs using various conditional statements and loops
4. Develop efficient, modular programs using functions
5. Utilize arrays, structures and unions for storing and manipulating data
6. Make use of files and file operations to store and retrieve data
7. Design applications using basic data structures like stacks, queues and linked lists.

INDEX

Introduction to Problem Solving

The computer allows us to do tasks more efficiently, quickly, and accurately than we could by hand. In order for this powerful machine to be a useful tool, it must be programmed. That is, we must specify what we want done and how. We do this through programming.

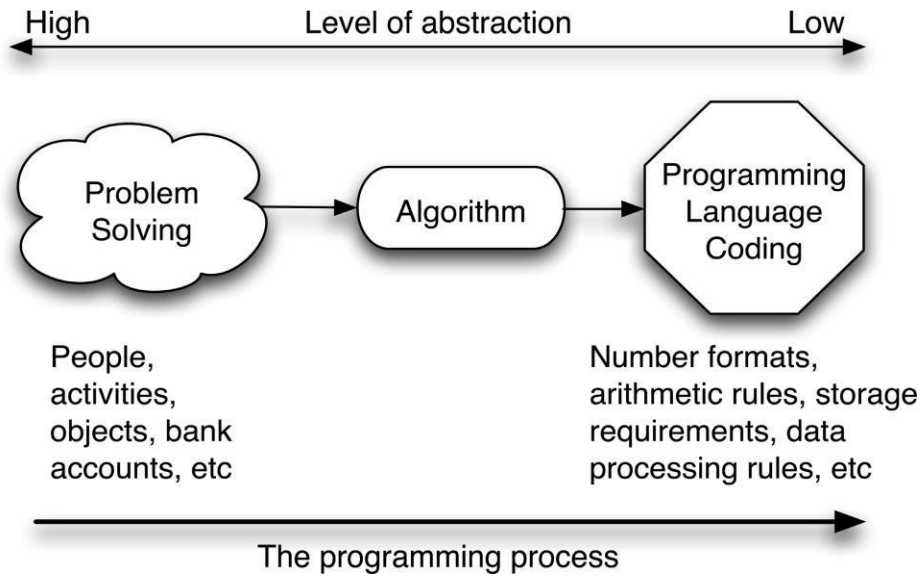
A computer is not intelligent. It cannot analyze a problem and come up with a solution. A human (the programmer) must analyze the problem, develop the instructions for solving the problem, and then have the computer carry out the instructions.

Once we have written a solution for the computer, the computer can repeat the solution very quickly and consistently, again and again. The computer frees people from repetitive and boring tasks.

Steps in developing a solution

Before starting to think about writing any programming code, we must first focus on the problem statement, that is, on the *real-world domain*. First we must *understand* the problem and then we try *solving* it. Once we think we have solved it we *systematize* our solution by writing it out in a more formal way as a series of steps (an *algorithm*) that can be followed by another person.

Problem solving requires thinking about the problem at a high level of abstraction while writing programming language code requires a very low level of abstraction



To write a program for a computer to follow, we must go through a two-phase process: problem solving and implementation

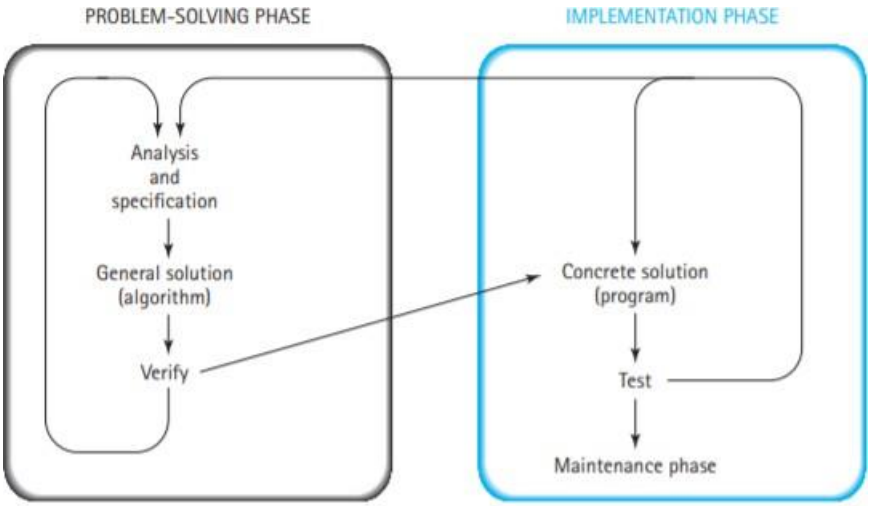


Figure 1.1 *Programming process*

Problem-Solving Phase

1. Analysis and Specification: Understand (define) the problem and what the solution must do.
2. General Solution (Algorithm): Specify the required data and the logical sequences of steps that solve the problem.
3. Verify: Follow the steps exactly to see if the solution really does solve the problem.

Implementation Phase

1. Concrete Solution (Program): Translate the algorithm (the general solution) into a programming language.
2. Test: Have the computer follow the instructions. Then manually check the results. If you find errors, analyze the program and the algorithm to determine the source of the errors, and then make corrections.

Once a program has been written, it enters a third phase: maintenance.

Maintenance Phase

1. Use: Use the program.
2. Maintain: Modify the program to meet changing requirements or to correct any errors that show up while using it.

The programmer begins the programming process by analyzing the problem, breaking it into manageable pieces, and developing a general solution for each piece called an algorithm. The solutions to the pieces are collected together to form a program that solves the original problem.

UNIT-I

Topics Covered:

Introduction to Computing – Computer Systems, Computing Environments, Computer Languages, Algorithms and Flowcharts, Steps for Creating and Running programs.

Introduction to C – History of C, Features of C, Structure of C Program, Character Set,

C Tokens - keywords, Identifiers, Constants, Data types, Variables. Operators, Expressions, Precedence and Associativity, Expression Evaluation, Type conversion, typedef, enum

Control Structures: Selection Statements(Decision Making) – if and switch statements, Repetition Statements (Loops) - while, for, do-while statements, Unconditional Statements – break, continue, goto, Command line arguments.

INTRODUCTION TO COMPUTING

Computer systems

Computer is a digital electronic device which understands only 0's and 1's. It performs all arithmetic calculations (Addition, multiplication...) and non arithmetic calculations (copy, choose, move, compare….).

A computer is a system made of two components: Hardware and Software

The computer hardware is the physical equipment. The software is the collection of programs (instructions) that allow the hardware to do its job. The computer manipulates these symbols in the desired way by following an intellectual map called program. A program is a detail set of humanly prepared instructions that directs the computer to function in a specific way to produce the desired results. Computer intelligence quotient or

I.Q is zero. It does not have any thinking, arguing or decision-taking power of its own. This power is intelligently conferred to it by proper programming methods by persons handling it.

Computer Hardware:

The hardware component of the computer system consists of five parts:

1. Input devices

2. Central processing unit (CPU)

3. Primary (Main) storage or Immediate Access memory Storage (I.A.S)

4. Output devices

5. Auxiliary storage devices or secondary memory or backing storage devices (floppy disk, tape...)

The input device is usually a keyboard where programs and data are entered into the computer. Examples of the other input devices include a mouse, a pen, or stylus, a touch screen, or an audio input unit.

The central processing unit (CPU) is responsible for executing instruction such as arithmetic calculations, comparisons among data, and movement of data inside the system.

Primary storage is a place where the programs and data are stored are erased when we turn off a personal computer or when we log off from a time-sharing computer.

The output device is usually a monitor or a printer to show output. If the output is shown on the monitor, we say we have a soft copy. If a printed on the printer, we say we have a hard copy.

Auxiliary storage is used for both input and output. It is a place where the programs and data are stored permanently. When we turn off the computer, our programs and data remain in the secondary storage, ready for the next time we need them.

Computer Software:

Computer software is divided into two categories: system software and application software. System software manages the computer resources. It provides the interface between the hardware and the users but does nothing to directly serve the users need. Application software is a directly responsible for helping users solves their problem.

System Software:

System software consists of programs that manages the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The operating system provides services such as a user interface, file and database access, and interface to communication systems such as internet protocols.

System support software provides system utilities and other operating services.

System development software includes the language translators that convert programs into machine language for execution, debugging tools to ensure that the programs are error free.

Application software:

These are divided into two classes:

General-purpose software and

Application-specific software

General-purpose software is purchased from a software developer and can be used for more than one application. Examples like word processor, database management systems.

Application-specific software can be used only for its intended purpose. A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples.

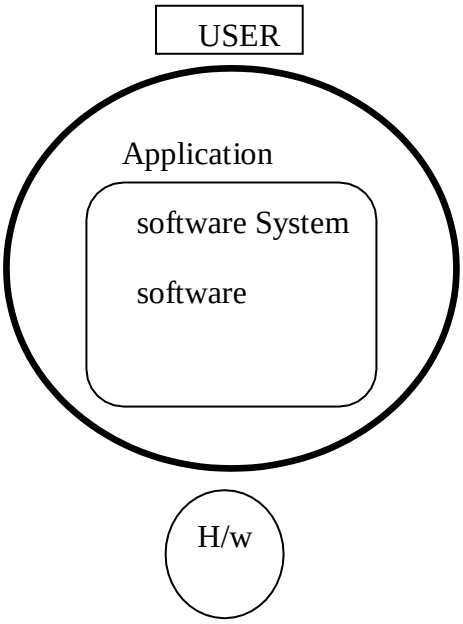


Figure: Relationship between System software and Application software

Computing Environments

In the early days of computer, there is only one environment: the mainframe computer hidden in a central computing department. With the advent of minicomputers and personal computer, the environment changed, resulting in computers on virtually every desktop.

Personal Computing Environment:

In 1971, Marcian E. Hoff, working for Intel, combined the basic elements of the central processing unit into the microprocessor. This first computer on a chip was Intel 4004 and was the grandparent many times removed of Intel's current system.

If we are using a personal computer, all of the computer hardware components are tied together in our personal computer(PC for short).we have whole the computer for ourselves and we can do whatever we want.

Time-Sharing Environment:

Employees in large companies often work in what is known as time-sharing environment. In the time-sharing environment, many users are connected to one or more computers. A typical college lab in which a minicomputer shared by many students. In the time-sharing environment, all computing must be done the central computer. In other words, the central computer has many duties: It must control the shared resources; it must manage the shared data and printing; and it must do the computing.

Client/Server Environment:

A client server computing environment splits the computing function between a central computer and user's computers. The users are given personal computers or workstation so that some of the computations responsibility can be moved from the central computer and assigned to the work stations. In this, the micro computers or work stations are called the client. The central computer, which may be a powerful microcomputer, minicomputer, or central main frame systems, is known as the server.

Distributed Computing:

A distributed computing environment provides a seamless integration of computing functions between different servers and clients. The Internet provides connectivity to different servers throughout the world. Example like eBay.

Computer languages

Low Level Language: The only programming language available was machine languages. Each computer has its own machine language which is made of streams of 0's and 1's. The only language understood by computer hardware is machine language.

1. Middle Level Language: The programs are written using the instructions of CPU is called Assembly level language. It uses symbols, or mnemonics, to represent the various machine language instructions. Examples are TASM,MASM

2. High Level Language: In the Assembly level language programmers need to concentrate on the hardware so it is tedious because each machine instruction had to be individually coded. So the High level language was developed it is English like language where instruction typically translates into machine language instructions.

Creating and Running Programs

The procedure for turning a program written in C into machine language. These steps are repeated many times during development to correct errors and make improvements to the code.

Writing and Editing Programs:

The software used to write programs is known as text editor. A text editor helps us enter, change, and store character data. Depending on the editor on our system, we could use it to write letters write programs. The main difference between the text processing and program writing is that program are written using lines of code, while most text processing is done with characters and lines. After complete a program, we save our file to disk. This file will be input to the compiler; it is known as a source file.

Compiling Programs:

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The C compiler is actually two separate programs: The Preprocessor and the translator

The preprocessor reads the source code and prepares it for translator. While preparing the code, it scans for special instruction known as preprocessor commands. These commands tell the preprocessor to look for the special code libraries, make substitutions in the code, and in other ways prepare the code for translation with machine language. The result of preprocessing is called the translation unit. After the preprocessor has prepared the code for compilation the translator does the actual work of converting the program into machine

language. An object module is the coding machine language.

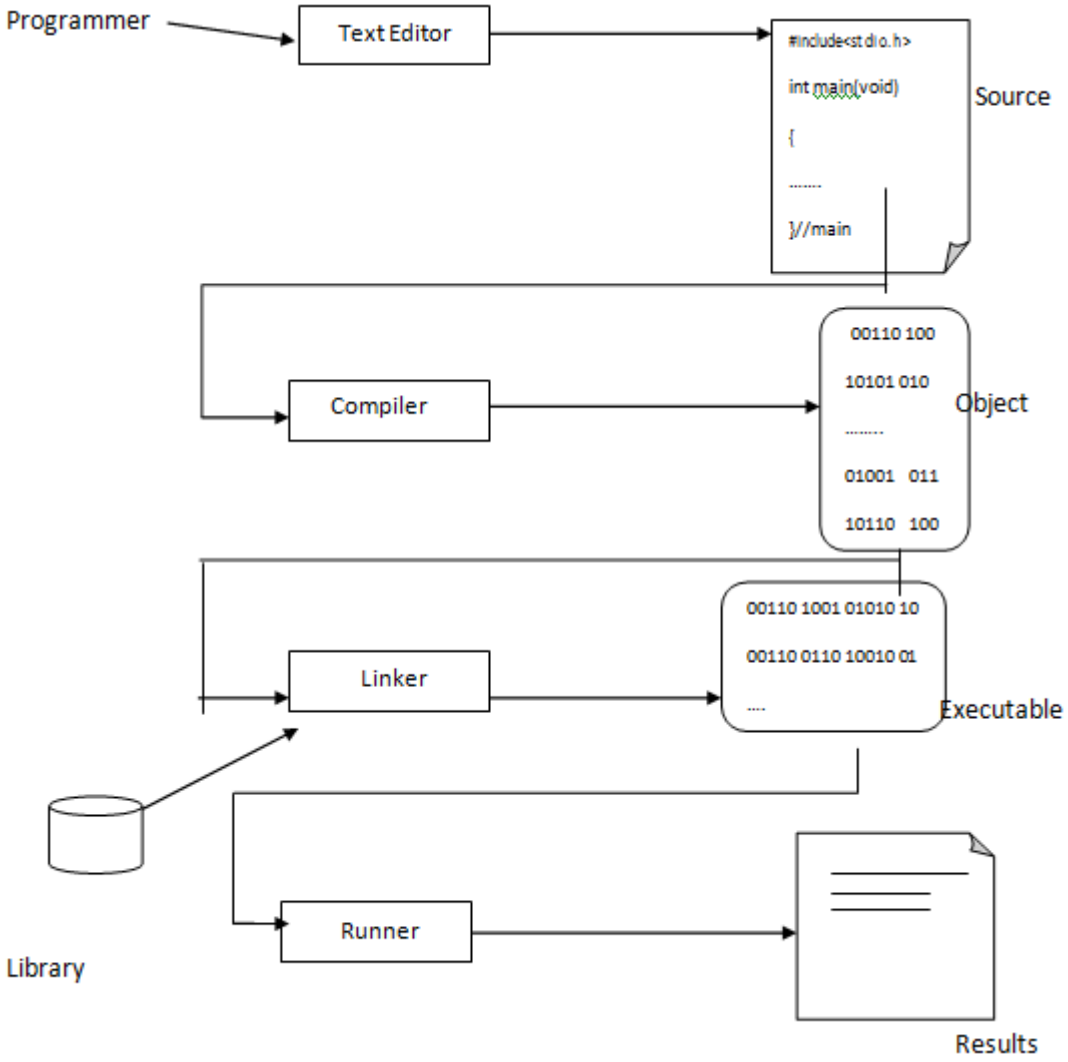


Figure: Building a C Program

Linking Programs:

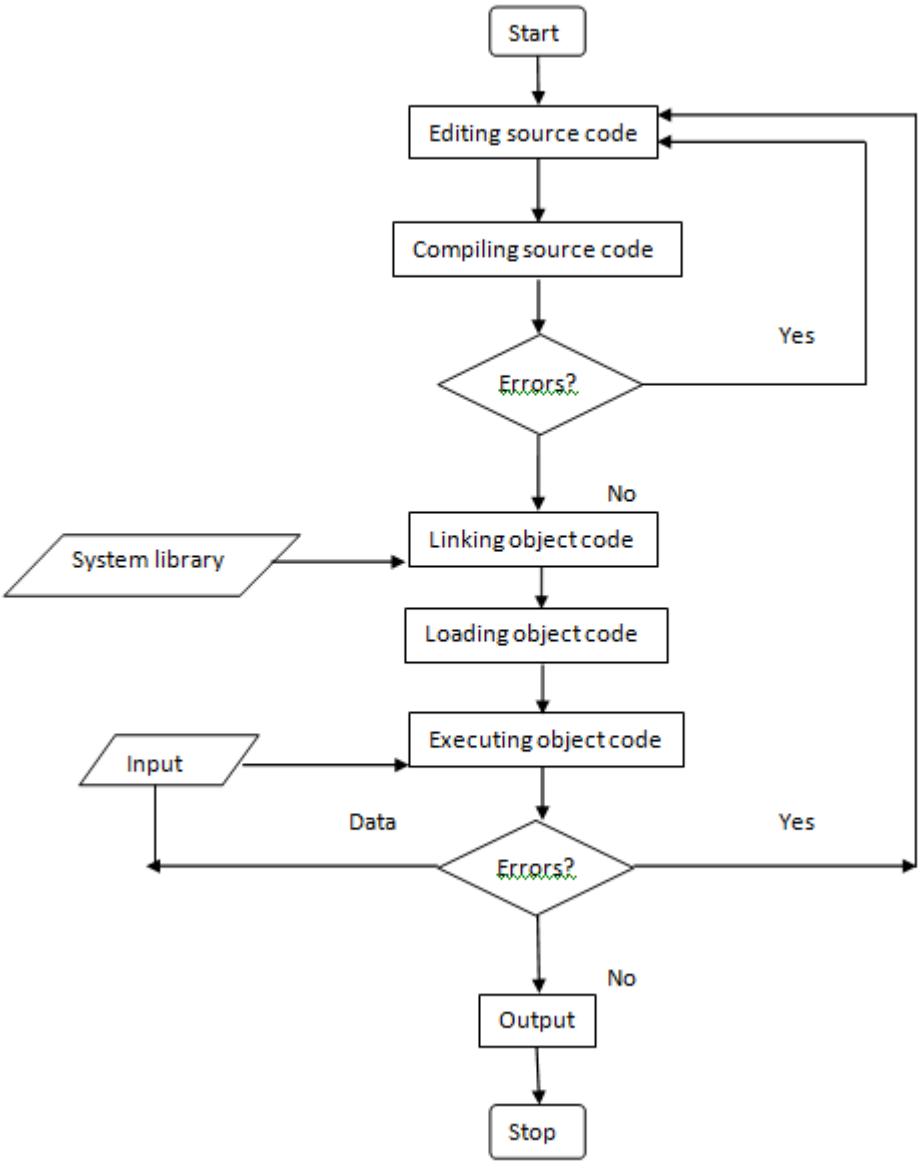
As we will see late, a C Program is made up of many functions. We write some of these functions, and they are a part of our source program. However, there are other functions, such as input/output process and mathematical library functions that exists else where and must be attached to our program. The linker all of these functions, ours and system's, into pour final executable programs.

Executing Programs:

Once our program has been linked, it is ready for execution. To execute a program, we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the loader. It locates the executable program and reads it into memory. When everything is loaded, the program takes control and it begins execution. In today's integrated development environment, these steps are combined under one mouse click or pull-down windows.

In typical program execution, the program reads data from processing, either from the user or from a file. After the program process the data, it prepares the output. Data output can be to the user monitor or to file. When the program has finished its job, it tells the operating system, which then removes the program from the memory.

Figure: Flow Chart for Process of Compiling and Running 'C' program



Algorithms

An algorithm is a step-by-step procedure for solving a problem using pseudocode. The Pseudocode we present here is particularly useful for developing algorithm that will be converted to structured C programs.

Features of an algorithm:

Finiteness : An algorithm terminates after a fixed number of steps.

Definiteness : Each step of the algorithm is precisely defined.

Effectiveness : All the operations used in the algorithm can be performed exactly in a fixed duration time.

Input : An algorithm has certain precise inputs before the execution of the algorithm begins.

Output : An algorithm has one or more outputs

Pseudo code

Pseudo code is similar to everyday English; Pseudo code is an artificial and informal language that helps programmers develops algorithms. It is convenient and user-friendly although it is not an actual computer programming language. Pseudo code programs are not actually executed on computers. Rather, they merely help the programmers "think out" a program before attempting to write it in a programming language such as C.

Steps to follow in writing pseudo code:

- Mimic good code and good English. Variable names should be mnemonic, include comments where it is useful.
- Ignore unnecessary details. Use some convention to group statements.(begin/end, brackets).
- Take advantage of programming short hands. Using if-else or looping structures is more concise.
- Consider the context.

Example: 1 An algorithm /Pseudo code to add two numbers.

```
Step 1:    start
Step 2:    Read the two numbers into a, b
Step 3:    c = a + b
Step 4:    write/print c
Step 5:    stop.
```

Example 2: An algorithm /Pseudo code to find whether a given number is odd number or a even number.

```
Step 1:    Start
Step 2:    Read the number n
Step 3:    If (n % 2) = 0 then
               Write n is even number Go to step 5
Step 4:    write n is odd number
Step 5:    Stop
```

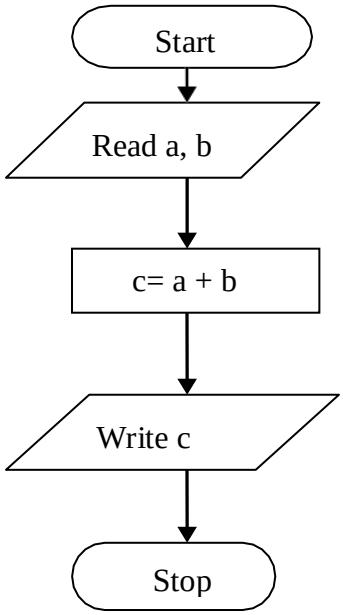
Flow Chart

A flow is a graphical representation of an algorithm or a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols such as rectangles, diamonds, ovals, and small circles as shown below. These symbols are connected by arrows called flowlines. Like pseudo code, flowchart is useful for developing and representing algorithms, although, pseudo code is performed by most programmers. Flowcharts clearly visually a how control structures operate in a program. The most common symbols used in drawing flow charts.

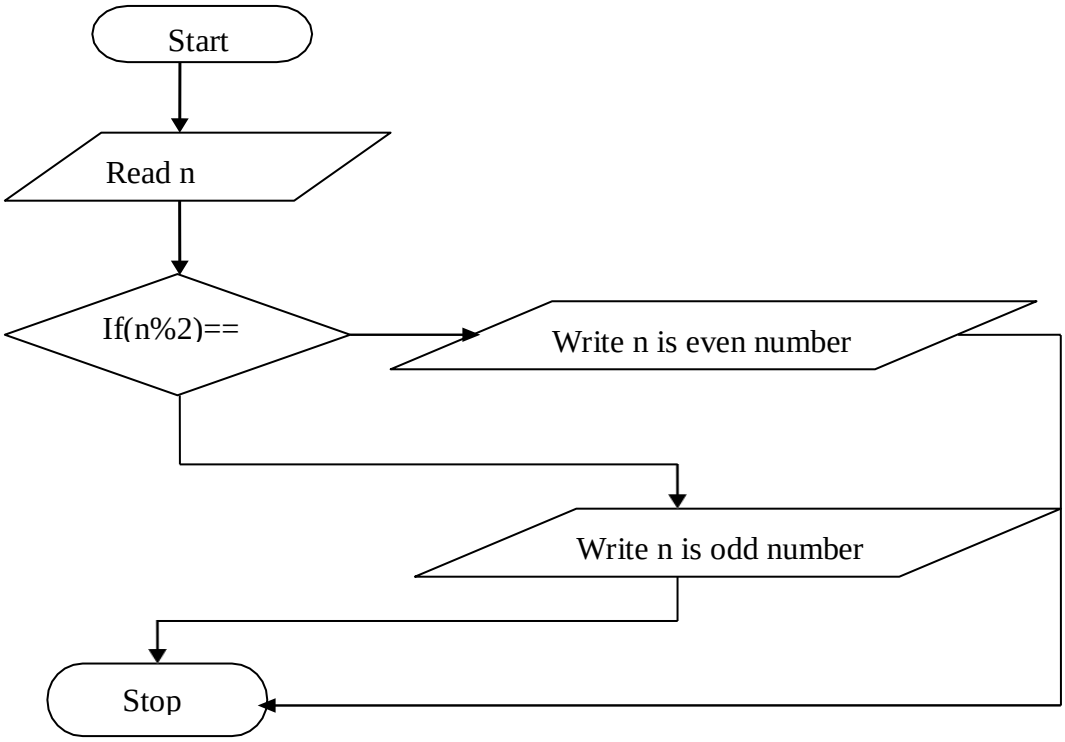
Uses of flowcharts:

- Flowcharts are an excellent means of communication and they impart ideas to others.
- Flowcharts provide an overview of the entire problem and its algorithm for solution.
- Flowcharts are quick method of illustrating program flow and show all major parts of a program.
- Flowcharts facilitates coding and modification of programs
- Flowcharts provide a permanent recording of program logic.

Example 1: Flowchart for addition of two numbers.



Example 2: Flowchart to find whether a given number is odd or even.



INTRODUCTION TO C LANGUAGE

History of C

C is a structured programming language. It is considered a high-level language because it allows the programmers to concentrate on the problem at hand and not worry about the machine that the program will be using. C, like modern languages, is derived from ALGOL. the first language to use a block structure. ALGOL is used in Europe but not in America this leads to the development of Structured programming languages. In 1967, Martin Richards developed a language called Basic Combined Programming Language, or BCPL. Ken Thompson followed in 1970 with similar language he simply called B. B was used to developed the first version of the UNIX, one of the popular network operating system in use today. Finally, Dennis M. Ritchie developed C, which took many concepts from ALGOL, BCPL, and B and added the concept of data types.

The history of 'C' started in 1972 by the BELL Laboratories, USA., Where Dennis M.Ritchie proposed this language. The growing popularity of 'C', the changes in the language over the years and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce "An unambiguous and machine- independent definition of the language C". The result is the ANSI standard for C.

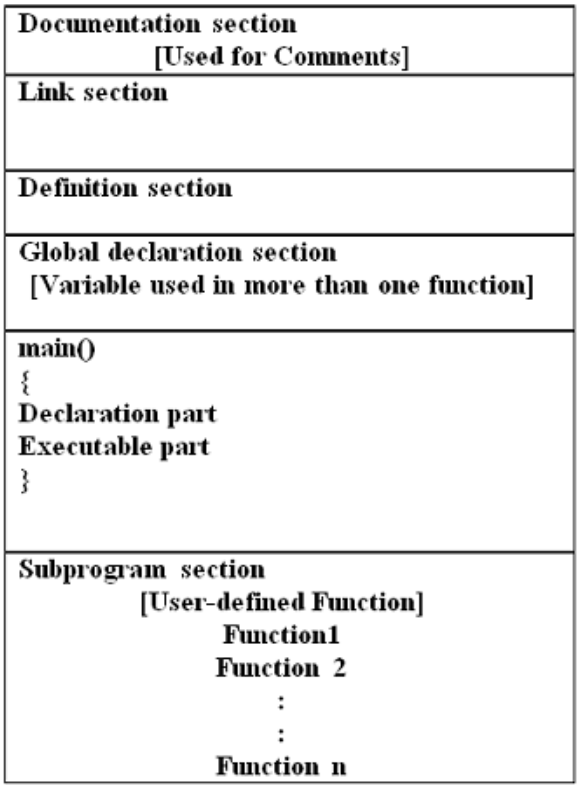
Structure of a C Program: Every C program contains a number of several building blocks known as functions. Each function of it performs task independently. A function is subroutine that may consists one or more statements.

Example of a simple C program:

```
/* Program Listing welcome.c A simple program in 'C'*/      line 1
#include<stdio.h>                                            line 2
 main()                                                      line 3
{                                                            line 4
 printf (" Welcome to C programming \n");                    line 5
} /*End of main () */                                       line 6
```

Output: Welcome to C Programming

BASIC STRUCTURE OF A 'C' PROGRAM:



Example:

```
//Sample Prog Created by:Bsource

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

void fun();

int a=10;

void main()
{
clrscr();
printf("a value inside main(): %d",a);
fun();
}

void fun()
{
printf("\na value inside fun(): %d",a);
}
```

Documentation section:

To enhance the readability of the program, programmers can provide comments about the program in this section. Comments can be used any where in the program but too many comments are avoided. It is useful for documentation. This gives the clarity of the program can be followed if it is properly documented. The comments can be inserted with a single statement or in nested statements.

Examples

/* This is a single comment*/

/*This is a example of/*nested comments*/*/

From above example: line 1

/* Program Listing welcome.c A simple program in 'C'*/

Header file section: C program depends upon some header files for function definition that are used in program. A header file contains the information required by the compiler when calls to the library functions used in the program occur during compilation. Each header file by default is extended with .h. The file should be included using #include directive.

Example: line 2

#include<stdio.h> or #include "stdio.h"

In this example <stdio.h> file is included i.e. all the definitions and prototypes of functions defined in this file are available in the current program. This file is compiled with original program.

Global declaration section: This section declares some variables that are used in more than one function. These variables are known as global variables. These must be declared outside of all the functions.

Main program section: Every program written in C language must contain main() function. Empty parenthesis after main is necessary. The function main() is starting point of every 'C' program. The execution of the program always begins with the function main(). Except the main () function, other sections may not be necessary. The program execution starts with the opening brace ({) and ends with the closing brace (}). Between these two braces the programmer should declare the declaration part and the executable part.

Declaration Part: the declaration part declares the entire variables that are used in executable part. The initialization of variables also done here. The initialization means providing the initial value to the variables

- **Executable Part:** This part contains the statements following the declaration of the variables. This part contains a set of statements. These statements are enclosed between the braces.

  Examples: line 3, line 4, line 5, line 6

User-defined function: The function defined by the user is called user defined function. These functions are generally defined after the main() function. They can also be defined before main() function. This portion in not compulsory.

Rules of a C language

- C is made of functions.
- A function is a set of statements which perform a specific task(job)
- Structure of a function definition: A function will have a name.
- A C statements ends with a semi-colon (;) which indicates to the compiler end of the statement.
- Most of the statements are placed with in a function with few exceptions.
- main () is a special type of function and its specialty is every program should contain one and only main function. main () indicates the starting point of program.

  i. A part from main() function. There are two types Standard Library functions: They are also called as ready maid function or built in function.

  Examples: printf(),scanf(),sqrt()….

  ii. User-defined functions: These functions have to be defined by the programmers.

- Average of compiler provide 500 functions
- C language is case-sensitive but not space-sensitive.
- ANSI C has 32 keywords called as reserve words

Note: Though main () is not a keyword it has to be written in lower case letter.

- A C statement is one of the following:
- An expression
  l.variable=R.value
  Example: z=x+y; x=5;
- A call to a function
  Example: printf (“Welcome”);
  y=sqrt(x);
- Combination of both the above i.e. it can be expression and a call to a function Example: z= x+sqrt(y);

Characteristics of 'C' language Or Features of 'C' language

1. Modularity
2. Extendability
3. Portability
4. Efficiency and Speed
5. Flexibility

Modularity: Ability to break down large modules into a manageable sub modules. It is one of the important features of the structured programming languages.

Example: calculator can be divided into arithmetic and trigonometric and again that is divided into sub modules.

Advantages:

- Different persons with different roles can be involved in developing the project.
- The project can be completed with in given period of time.
- Debugging will become easier.
- Software maintenance become easier because of plug-out and plug in features.
- C is highly structured language.

Extendability: Ability to extend an existing software by adding new features is called extendabiltiy. It is a by product of modularity.

Portability: Ability to port (install) existing software in different platforms (Unix, dos etc.) is called portability. It means that the programs written on one machine can be executed on different machine with or without minor changes in the program.

Efficiency and speed: C language has rich set of data types and operators making the language

more efficient and fast. The language provides some operators example: increment and decrement operators) which speed up the execution to large extent. Further, it also provides user defined data types (example: structures) through which miscellaneous data can be easily manipulated.

Note: Though C is a high level language often it is called as middle level language because programs written in c language run at the speeds matching to that of the some programs written in assembly level language. Due to this reason C language is widely used in developing system software.

Flexibility: keywords are for computer languages. ASCI C complier has only 32 keywords. These are the basis building blocks of the programming language. C is also called as programmers' language. With 32 keywords a programmer is able to solve any type of task due to this reason programmers have the complete control over the language their by gaining the flexibility.

Note: C program run faster and occupy less space, they are widely used in mobile technology, telecommunications, embedded systems.

C Tokens:

In a Passage of text, individual words and punctuation marks are called as tokens. In a C program, the smallest individual units are known as C tokens. The following are different categories of C tokens.

1) Keywords 2) Identifiers 3) Constants 4) Operators.

Keywords:

Keywords are also known as reserve words. These keywords are only to be used for their intended purpose and as identifiers. All the Keywords are lowercase.

Example: auto, char, int, long, goto, return, register…etc

Note: The keywords with at least one uppercase letter can be used as an identifier.

Identifiers:

Identifier allows us to name data and other objects in the program. Different programming languages use different syntactical rules to form identifiers. In C, the rules for identifiers are very simple. The only valid names symbols are the capital letter A through Z, the lowercase letters a through z, the digits 0 and 9, and the underscore. The first character of the identifiers cannot be a digit. The last rule is that the name we create cannot be keywords. C allows names to be up to 63 characters long.

Rules of identifiers

1. An identifier must start with a letter or underscore: it may not have a space or a hypen.
2. C is a case-sensitive language

Variables

A variable is a place holder that occupies some part of the memory which can initialize or assigned or changed during the execution of the program.

Declaration of variables

Syntax: datatype variable _name(s);

Example**:** float price;

Here price is a variable which hold a float value.

Initialization of variable

Syntax: data type variable_name(s) = value;

Example: int x=5;

Here x is a variable which holds a value=5;

Rules & conventions for naming the variables:

1. Variable should always be declared at the beginning of the functions.
2. A variable name should begin with alphabet or underscore. And the remaining character can be alphabets digits or underscore.
3. A variable name can't exceed 31 characters
4. Special characters are not allowed with in the variable name including spaces. Expect underscore.
5. Keywords cannot be used as variable names

Conventions

1. Declare variable names in lowercase letters
2. Variables names should provide a hint on what it is going to hold.

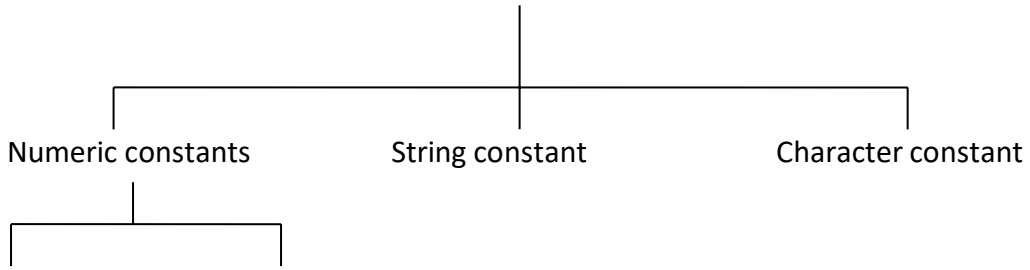
Examples of valid and invalid variables:

Valid variables	Invalid variables
int a2; int _a1; double a_b_c;	float 1a; int a b c; int abc$; double $abc;

Note: Uninitialized variables contain junk or garbage values.

Constants A constant is a value that doesn't change during the execution of a program. These are generally declared before main function.

Constants



Integer constant: Numeric value without decimal part is called integer constant.

Example: 5, -66

Real constant: Numeric value with decimal part is called as real constant. The precision of a real value indicates number of digits after decimal point. Example: 5.0, - 98.76…

Character constant: A single character enclosed in single quotes is called as character constant.

Character Set
Alphabet: a-z A-Z Digits: 0-9 Special characters: $, ?, ; , , …..

Example: ‘a’, ‘5’,

Declaration of a constant

Syntax: const data type constant_name= value;

Example: const float PI=3.14159;

Note: In an 8 bit character set, a character occupies 1 byte. Where as in 16 bit character set, a character occupies 2 bytes. Character constants have integer values known as ASCII values

(ASCII-American Standard code for Information Interchange).Since each character represents an integer value, it is also possible to perform arithmetic operations on character constants. Spacebar is also a character constant.

String constant: A single character or a group of characters enclosed in double quotes is called string constant.

Example: “5” , “temperature”

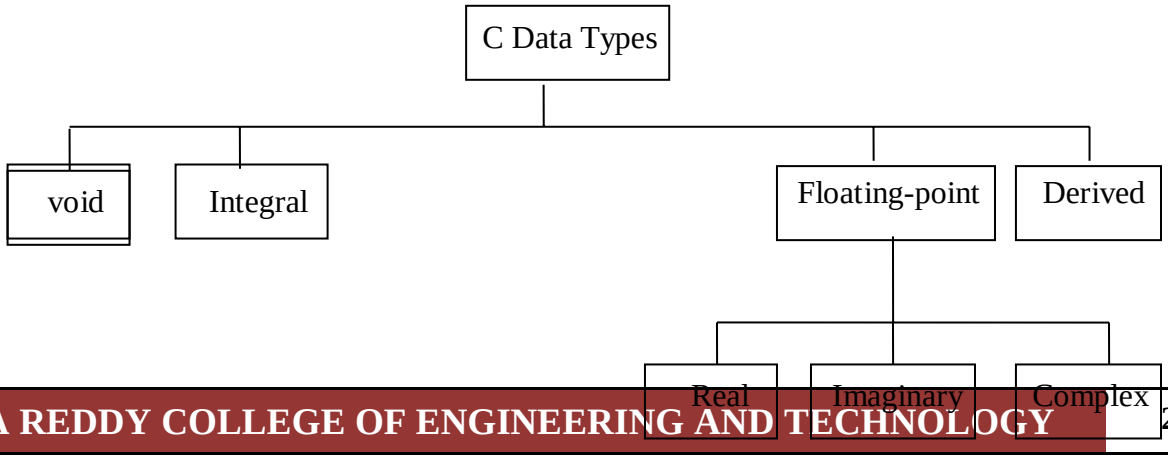
Note: Character constant occupies 1 byte where as string constant occupies 2 bytes.

Backslash Character constants: C supports some special backslash character constants that are used in output functions. These are also known as ***escape sequences***.

‘\a’- audible alert(bell)	‘\v’- vertical tab
‘\b’- backspace.	‘\’’- single quote.
‘\f’- form feed	‘\”’- double quote.
‘\n’- new line	‘\?’- question mark
‘\r’- carriage return	‘\0’- Null

Data Types

A data type consists of the values it represents and the operations defined upon it. The C language has defined a set of data types that can be divided into four general categories: void, integral, floating-point, and derived as shown below



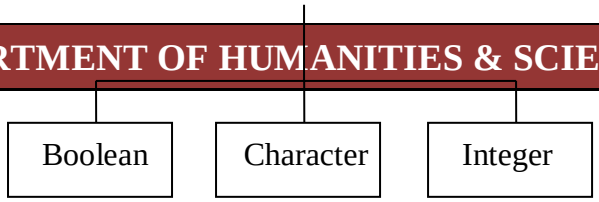


Figure: Data Types

Void type:

The void type, designated by the keyword void, has no values and no operations, although having no values and operations might seem unusual. It can also be used to define that a function has no return value.

Integral type

The C language has three integral types: Boolean, character, and integer. Integral types cannot contain a fraction part; they are whole numbers.

- **Boolean:** Boolean type can represent only two values: true or false. C used integers to represent the Boolean values: a nonzero number (Positive or negative) was used to represent true, and zero was used to represent false. The Boolean type, which is referred to the keyword bool, is stored in memory as 0(false) or 1(true).
- **Character:** A character is any value that can be represented in the computer's alphabet, or as it is better known, its character set. An 8-bit character set represent up to 256 characters.
- **Integer:** An integer type is a number without a fraction part. C supports four different sizes of the integer data types:
- **Floating-Point Types** The C standard recognizes three floating –point types: real, imaginary, and complex.
- **Real:** The real type holds values that consist of an integral and a fractional part. The C language supports three different sizes of real types: float, double, and long double.

- **Imaginary type:** An imaginary number is used extensively in mathematics and engineering. An imaginary number is a real number multiplied by the square root of -1. Most C implementations do not support the imaginary type.

- **Complex:** Complex number is a combination of real and imaginary number.

DATA TYPE	STORAGE SIZE IN BYTES		RANGE		FORMAT SPECIFIER
	16 BIT Compiler	32 BIT Compiler	16 BIT	32 BIT	
short int or signed short int	2	2	-32768 To 32767 (-2^{15} to $+2^{15}$ -1)	-32768 To 32767 (-2^{15} to $+2^{15}$ -1)	%hd
short int or signed short int	2	2	0 to 65535 (0 to $+2^{16}$ -1)	0 to 65535 (0 to $+2^{16}$ -1)	%hu
signed int or int	2	4	-32768 To 32767 (-2^{15} to $+2^{15}$ -1)	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}$-1)	%d or %i
unsigned int	2	4	0 to 65535 (0 to $+2^{16}$ -1)	0 to 4,294,967,295 (0 to2^{32}-1)	%u
long int or signed long int	4	4	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}$-1)	-2,147,843,648 to 2,147,843,647 (-2^{31} to $+2^{31}$-1)	%ld
unsigned long int	4	4	0 to 4,294,967,295 (0 to2^{32}-1)	0 to 4,294,967,295 (0 to2^{32}-1)	%lu
long long int or signed long long int	Not supported	8	--------	-9223372036854775808 To 9223372036854775807 (-2^{63} to $+2^{63}$-1)	%Ld
char or signed char	1	1	-128 to 127 (-2^{7} to 2^{7} -1)	-128 to 127 (-2^{7} to 2^{7} -1)	%c
Unsigned signed char	1	1	0 to 256 (0 to 2^{8}-1)	0 to 256 (0 to 2^{8}-1)	%c

Size and range of floating point data type is shown in the table:

Data type (key word)	Size (memory)	Range	format specifier
Float	32 bits (4 bytes)	3.4E-38 to 3.4E+38	%f
Double	64 bits (8 bytes)	1.7E-308 to 1.7E +308	%lf
long double	80 bits (10 bytes)	3.4E-4932 to 1.1E+4932	%Lf

Derived data types:

Derived datatypes are used in 'C' to store a set of data values. Arrays , Structures , Union and pointer are examples for derived data types.

User-defined data types:

The data types defined by the user are known as the user-defined data types. C provides two identifiers ***typedef*** and ***enum*** to create new data type names

typedef:

It allows the user to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables.

Syntax: typedef type identifier;

For Example, the declaration ,

typedef int Integer;

makes the name Integer a synonym of int.Now the type Integer can be used in declarations ,type castings,etc like

Integer num1,num2;

Which will be treated by the C compiler as the declaration of num1,num2as int variables.

"typedef" ia more useful with structures and pointers.

Enumeration:

An enumerated type (also called enumeration or enum) is a data type consisting of a set of named values called elements, members or enumerators of the type.

The enumerator names are usually identifiers that behave as constants in the language. A

variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. In other words, an enumerated type has values that are different from each other, and that can be compared and assigned, but which are not specified by the programmer as having any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily.

The enumeration data type is defined as follows:

Syntax: enum identifier {value1,value2,……valuen};

Identifier: it 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 (*enumeration constants*).

Declaration of variables of this new type:

Ex:

enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};

declaration of variable:

enum week today;

today=wednesday;

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday}; int
main()
{
   enum week today;
   today=wednesday;
   printf("%d day",today+1);
   return 0;
  }
```

Output

4 day

Operators:

Operators are C tokens which can join together individual constants, variables, array elements and functions references. Operators act upon data items called operators.

C operators are classified into following categories:

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

Arithmetic operators:

Arithmetic operators	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

- For arithmetic operations, operands must be numeric values.
- Here modulo division produces remainder of integer division

Arithmetic operations are classified as

- Integer arithmetic
- Real arithmetic
- Mixed mode arithmetic

Integer arithmetic: Both the operands are integer in an integer arithmetic. It always yields an integer value.

Example: a+b, a-b

Real arithmetic: Both the operands are real in real arithmetic. It yields real value as result. It cannot be applied to % operator.

Example: float a=20.0, b= 3.0
a+b= 20.0+3.0=23.000000

a/b=20.0/3.0=6.6666667

a%b=20.0%3.0(invalid expression)

Mixed mode arithmetic: when one is integer and other is real it is known as mixed mode arithmetic. Here, result will always be real.

Example: int a=20.0, b=3;

a+b= 20.0+3=23.0

a/b=20.0/3=6.666667

a%b=20.0%3 (invalid expression)

Relational operators:

Relational operators	Action
<	Is less than
<=	Is less than or equal to
>	Is grater than
>=	Is grater than or equal to
==	Is equal to
!=	Is not equal to

Example

1) 38<4 = 0 false

2) 38>4 =1 true

Logical operators:

Logical operators	Action
&&	Logical AND
||	Logical OR
!	Logical NOT

- Logical expression combines two or more relational expressions.
- Logical operators are used to test more than one condition and make decision.

Logical AND: The result of the logical AND expression will be true only when the relational expressions are true.

Syntax: Exp1 && Exp2

Example: if a=10, b=5, c=15

1. i=(a>b)&&(b<c); The value of i in this expression will be 1.
2. i=(a<b)&&(b<c); The value of i in this expression will be 0.

Logical OR: The result of logical OR expression will be true only when both the relational expressions are false.

Syntax: Exp1||Exp2 Example:

if a=10,b=5,c=15

1. i=(a<b)||(b<c); The value of i in this expression will be 1.
2. i=(a<b)||(b>c); The value of i in this expression will be 0.

Logical NOT: the result of the expression will be true, if the expression is false and vice versa.

Syntax: !Exp2

Example: x=20

i = !(x==20) The value of i will be 0. This is because x==20 is true(value 1).

Assignment operators:

The operators are used to assign result of an expression to a variable. Syntax:

variable op=exp; OR variable=variable op exp;

Example:

1. a=a+10; or a+=10; adds 10 to a and assigns the value to a.
2. a=a*(b+5) or a*=b+5 multiplies b+5 with a and assigns result to a

Increment and Decrement operators: These operators are represented as '++' and '--'. '++' increments operand by 1 and '--' decrements operand by 1. They are unary operators and take the following form.

Examples:

1. int a=9;
   y= ++a; The value of y will be 10;

2. int a=9;

   y=a++; The value of y will be 9;

Operator	Action
a++	Post-increment
++a	Pre-increment
a- -	Post-decrement
--a	Pre-decrement

Conditional operator: It is also known as ternary operator.

Syantax: Exp1 ? Exp2 : Exp3; where exp1,exp2,exp3 are expressions. Example:

int a=5,b=10,c=15; y =

(a>b) ? b : c;

In the above statement, the expression a>b is evaluated since; it is false value of c will be assigned to y. so value of y will be 15.

Bitwise operators:

Bitwise operators are similar to that of logical operators except that they work on binary bits. When bitwise operators are used with variables, they are internally converted to binary numbers and then bitwise operators are applied on individual bits. Bitwise operators do manipulation on bits stored in memory. These operators work with char and int data types. They cannot be used with floating point numbers.

Bitwise operators	Action
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right
~ (unary)	Ones complement

Example: if a=4, b=6;

The equivalent binary value of a is 0000 0100 The

equivalent binary value of b is 0000 0110

a) The value of a&b = 0 0 0 0 0 1 0 0

b) The value of a|b= 0 0 0 0 0 1 1 0

c) The value of a^b = 0 0 0 0 0 1 0 1

d) a<<2. It left shifts the binary bits twice. Therefore a= 0 0 0 1 0 0 0 0

e) a>>2. It right shifts the binary bits twice. Therefore a = 0 0 0 0 0 0 0 1

f) ~a. Unary operator which performs logical negation on each bit, 0's will be converted to 1's and 1's to 0's. Therefore a will become 1 1 1 1 1 0 1 1

Special operators:

There are some special operators available in C such as,

- Comma operator
- Sizeof operator
- Member selection operators (. and ->)

Comma operator: The comma operator ' , ' has the lowest priority among all the operators available in C. This operator is used to separate the related expressions. The expressions separated by comma operator need not be included within the parenthesis.

Example: c= (a=5, b=10, a+b);

This expression first assign 5 to a, then assigns 10 to b and finally (5+10) to c.

Sizeof operator: It returns number of bytes the operand occupies. Operand may be a variable, a constant, or a data type qualifier.

Syntax: sizeof (operand);

Example: y= sizeof (int); here value of y will be 2 bytes

Member operators: These are used to access members of structures and unions.

Example:

Var.member1; // when var is a structures variable.

Var->members2; // when var is a pointer to structures variable.

Points to be remembered on operators:

- Unary operator: It requires only one operand.
- Binary operator: It requires two operands like a+b, a-b,a*c etc.
- Logical operator: Used to compare two or more operands. Operands may be variable constants or expression.
- Assignment operator: Used to combine the results of two or more statements.
- Conditional operator: Checks the condition and executes the statement depending on the condition.
- Bitwise operator: Used to manipulate the data at bit level. It operates on integer only. It may not be applied to float or real.

Input/output Functions

Input and output operations on data can be done by using input and output functions. An input function reads data from keyboard and stores in the variable where as output function prints value in the variable on the screen

These functions are classified into two categories.

1. Formatted input/output functions(IO)
2. Unformatted input/output functions(IO)

FORMATTED I/O FUNCTIONS

•Formatted functions read and write all types of data values. format specifier are used to identify the data type to read or written into variables.

Formatted Input function:

Formatted input function accepts an input data that has been arranged in a particular format through keyboard.

scanf() is the formatted input function

The general form or syntax of the scanf( ) is

scanf("format specifier",&variable);

scanf function consists of two parts.

first part in scanf function is format specifier enclosed within double quotes. format specifier is used to specify the type of format accepted by the scanf to the specified variable. format specifier starts with "%" followed on or two characters

ex: %d is for accepting signed integers

Second part in scanf () is list of variables. All variables are preceded with an ampersand(&) symbol. ampersand is address operator which directs scanf() to store read value at the memory address of the variable.

format string(Control string): Group of format specifiers are called as format string or control string. if scanf() has to read multiple values from keyboard then all format specifiers are grouped to form format string as show below

ex: scanf("%d%%d%d",&v1,&v2,&v3);

The above example reads three integers one by one from keyboard into variable v1,v2,v3 respectively

Formatted Output function: This prints data on the console(output screen).

printf() is the formatted output function.

printf ( ): This function prints all type of data values to the console.

it consists of two sections. first section contains format string and next part contains list for corresponding variables for specified format string.

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

The control string consists of

- Characters that will be printed on the screen as they appear.
- Format specifiers that define the output format for display each data item.
- Escape sequence characters such as \t, \n, \b etc.

arg1,arg2…..argn are the list of variables whose values has to be printer on to screen

Note: The arguments should match in number, order and type with the format specifications.

Output of Integer Numbers:

List of format specifiers:

Format specifier	Data type
%hd	signed short int Or short int or short
%hu	unsigned short int
%d or %i	signed int Or int
%u	unsigned int
%ld	long int or signed long int
%lu	unsigned long int
%Ld	long long int Or signed long int
%Lu	unsigned long long int
%o or %O	octal integer
%x or %X	Hexa decimal integer
%c	char
%f	float
%lf	double
%Lf	long double
%s	String(group of characters)
%e	Print float value in exponential form.
%g	Print using %e or %f whichever is smaller
%e	Print float value in exponential form

The format specifier can also contain sub-specifiers: flags, width, .precision and code

%	FLAG	WIDTH MODIFIER	Precision	SIZE	CODE

flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.

width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

.precision	**Description**
.	Separates the fractional part. If the period is specified without an explicit value for *precision*, 0 is assumed.
Size	Allocates no of positions for fractional part

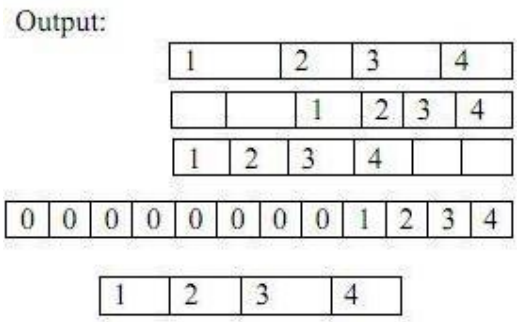
CODE using conversion symbol for data type

Ex: Output of the number 1234.

```
#include<stdio.h>
void main ( )
{
printf("\n%d",1234); //Without field width
printf("\n%6d",1234); // Right alignment
printf("\n%-6d ",1234); //Left alignment
printf("\n%012d",1234); //Padding with Zero
```

```
printf("\n%2d",1234); //Overriding the field width value.
}
```

Output:



For floating point **l Number:**

The output of a real number may be displayed in two ways:

1. Decimal Notation ex: 2.345, 0.000005, -2.8976, 3456.987 etc.
2. Exponential Notation Output of Real Number in decimal notation: %w.p f Here both ***w*** and ***p*** are integers.

The integer ***w*** indicates the minimum number of positions that are to be used for display of the value.

The integer ***p*** indicates the number of digits to be displayed after decimal point (precision). The value when displayed is rounded to p decimal places and printed right-justified in

```
#include<stdio.h> int
main()
{
        float y= 98.7654;
        printf("\n %f", y);
        printf("\n%7.4f",y);
        printf("\n%7.2f",y);
        printf("\n%-7.2f",y);
        return 0;
}
```

9	8	.	7	6	5	4
9	8	.	7	6	5	4
		9	8	.	7	7
9	8	.	7	7		

Printing of a single character:

The format specification for printing a character is:

```
#include<stdio.h>
int main()
{
printf("%c",'S');
printf("\n%3c",'S');
printf("\n%6c",'S');
        return 0;
}
```

S

S

S

UNFORMATTED IO FUNCTIONS

INPUT FUNCTIONS

1.getch()

2.getche()

3.getchar()

4.gets()

1. getch():- reads a character from keyboard and it does not require pressing of enter key after entering a key, entered character is not displayed on console

```
char ch;
ch=getch();
```

2. getche():- reads a character from keyboard and it does not require pressing of enter key after entering a key, entered character is echoed back on to the console

```
char ch;
ch=getche();
```

getch() and getche() are available in the header file conio.h (console input output header file)

3. getchar():-reads a character from keyboard and it requires pressing of enter key

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

4. gets():-reads a string from keyboard

```
char ch[10]; gets(ch);
```

OUTPUT FUNCTIONS:

1.putchar()

2.puts()

3.putch()

1. putchar():-prints a character on to the screen

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

2. puts():- prints a string on to the screen

```
char ch[10];
gets();
puts(ch);
```

3.putch():-prints a character on to the screen

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

Expressions

An expression consists of a single entity such as a constant, a variable, an array element or a reference to a function or can be a combination of such entities joined together using one or more operators. An expression can also represent logical condition, that is true or false, which can be represented using integer value like '1' and '0' respectively. Expressions can be simple or complex. An operator is a syntactical token that requires an action be taken. An operand is an object on which an operation is performed;

A simple expression contains only one operator.

Example: 2+5=7;

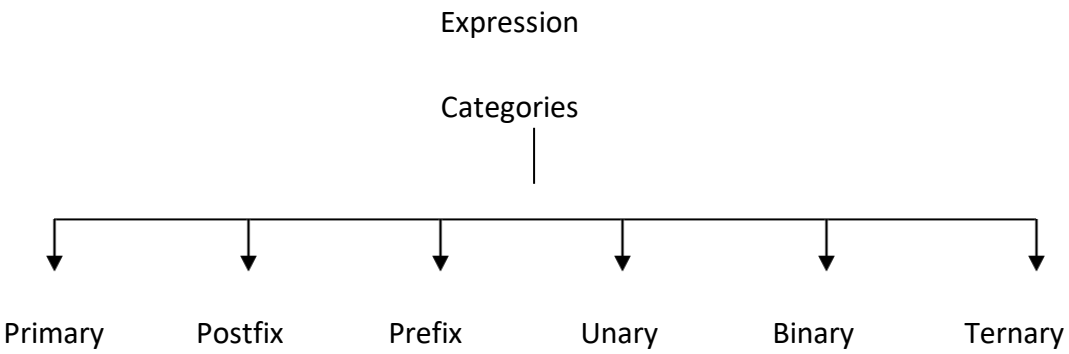
A complex expression contains more than one operator. Example:

2+6*8=50

Here a priority comes into picture with precedence. The order in which the operators in a complex expression are evaluated is determined by a set of priorities known as precedence. Higher the precedence that expression will be calculated first. If two operators with the same precedence occur in a complex expression. Another attribute of an operator, its associativity, takes control. Associativity is parsing direction used to evaluate an expression. It can be either left-to-right or right-to-left.

Note: when two operators having same precedence occur in an expression and their associativity is left-to-right.

We can divide the simple expression into six categories.



Examples:

1. c=a+b*d; 2.

   x==y;

   3. x+=y;

Note: An expression always reduces to single value.

Primary Expressions: A primary expression consists of only one operand with no operator. In c, the operand in the primary expression can be name, a constant, or a parenthesized expression.

Names: A name is any identifier for a variable, a function or any other object in the language. The following are examples of some names used as primary expressions:

Examples: a, b12, price, calc, INT_MAX, size etc.

Literal constants: A constant is a piece of data whose value can't change during the execution of the program.

Examples: 5, 123.98, 'A'

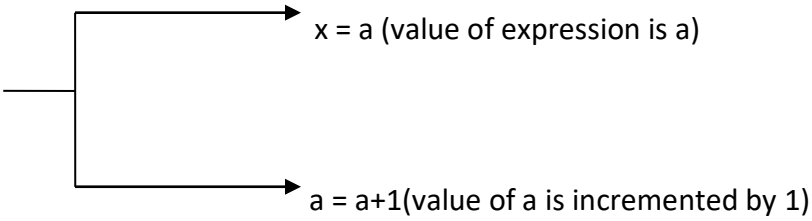
Parenthetical expressions: Any value enclosed in parentheses must be reducible to a single value and is therefore a primary expression.

Examples: (2+3+4), (a=23+b*6)

Postfix Expressions: The postfix expression consists of one operand following by one operator.

Postfix increments/decrements: The postfix increments and postfix decrements are also postfix operators. The value of the postfix increment expression is determined before the variable is increased. For instance, if the variable a contains 4 before the expression is evaluated, the value of the expression a++ is 4. As a result of evaluating the expression and its side effect, a contains 5.

Example: x=a++

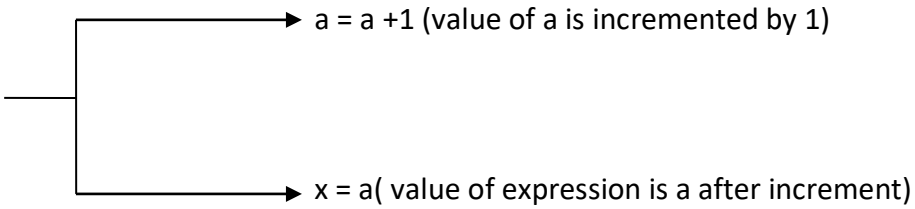


Note: The operand in a postfix expression must be a variable.

Prefix Expressions: The operator comes before the operand.

Prefix increment/decrement: In C, we have only two prefix operators that form prefix expressions: prefix increment and prefix decrement. With the prefix operators, the effect takes place before the expression that contains the operator is evaluated.

Example: x=++a



if the variable a contains 4 before the expression is evaluated, after evaluation the value of both a and x will be 5.

Unary Expressions: A unary expression, like a prefix expression, consists of one operand and one operator. The major difference is a prefix expression needs a variable while unary expression can have an expression or a variable as the operand.

Sizeof: The sizeof operator tells us the size, in bytes, of a type or a primary expression. On some PC the size of int is 2 bytes, some mainframes it is 4 bytes, and on super computers it is 16 bytes. It is important to know the exact size of an integer; we can use the sizeof operator with the integer type.

Note: It is also possible to find the size of primary expression.

Examples: sizeof -345.23, sizeof x;

Unary plus/minus: The unary plus and unary minus are what we think of as simply the plus and minus signs. In C, however, they are actually operators. Because they are operators, they can be used to compute the arithmetic value of an operand.

Expression	Contents of a Before expression	Expression value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

Binary Expressions: These binary expressions are formed by an operand – operator- operand combination.

Examples: Any two numbers added, subtracted, multiplied or divided are usually formed in algebraic notation, which is a binary expression.

Assignment expression: The assignment expression evaluates the operand on the right side of the operator (=) and places its value in the variable on the left. The assignment expression has a value and a side effect.

- The value of the total expression is the value of the expression on the right of the assignment operator (=).

- The side effect places the expression value in the variable on the left of the assignment operator.

Note: The left operand in an assignment expression must be a single variable.

Simple assignment: a = 5, b =x+1, i = i+1

The left variable must be able to receive it; that is, it must be a variable, not a constant. If the left operand cannot receive a value and we assign one to it,we get a compile error.

Compound assignment: A compound assignment is a shorthand notation for a simple assignment. It requires that the left operand be repeated as a part of the right expression. Five compound assignment operators are *=, +=, /=, %=, -=.

Example: x*=y+3 is evaluated as x = x*(y+3)

Precedence and Associativity

Precedence is used to determine the order in which different operators in a complex expression are evaluated. Associativity is used to determine the order in which operators with the same precedence are evaluated in a complex expression.

Precedence:

Example: 2+3*7

This expression is actually two binary expressions, with one addition and on multiplication operators. So the multiplication has the higher priority then 3*7=21 and adds the remaining 21+2=23.

Associativity: Associativity can be left-to-right or right-to-left. Left-to-right associativity evaluates the expression by starting on the left and moving to the right. Right-to-left associativity evaluates the expression proceeding from right to left.

Note: Associativity is applied when we have more than one operator of the same precedence level in an expression.

Left-to-right associativity:

Example: 3 * 8 / 4 % 4 * 5

Here all these operators have the same precedence so associativity from left to right.

((((3 * 8) / 4) % 4) * 5) → (((24/4) %4) *5) → ((6%4) * 5) → (2 *5) → 10

Right- to-left associativity:

Example: a +=b*=c-= 5

Which is evaluated as (a+=(b*=(c-=5)))

Again is expanded to (a=a+(b=b*(c=c-5)))

If a=3,b=5,c=8

(a=3+(b=(5*(c=8-5))) → then c=3,b=15,a=18.

PRIORITY AND ASSOCIATIVITY OPERATORS:

Operators	Description	Associativity	Priority/precedence
( )	Function call/ Parenthesis	Left to right	1
[ ]	Array expression		
->	Structure operator		
.	Structure operator		
+	Unary plus	Right to left	2
-	Unary minus		
++	Increment		
--	Decrement		
!	NOT operator		
~	Ones complement		
*	Pointer operator		
&	Address operator		
Sizeof	Size of operator		
Type	Type casting operator		
*	Multiplication	Left to right	3
/	Division		
%	Mod		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
? :	Conditional operator	Right to left	13
=,+=,*=,/=,%=,&=,|=, <<=,>>=,^=,-=	Assignment operator	Right to left	14
,	Comma operator	Left to right	15

Side Effects: In assignment C first evaluates the expression on the right of the assignment operator and then places the value in the left variable. Changing the value of the left variable is a side effect.

Example: x =4;

This simple expression has three parts. First, on the right of the assignment operator is a primary expression that has the value 4. Second, the whole expression (x=4) also has a value of 4. And third, as a side effect, x receives the value 4.

Let's modify the expression slightly

x = x+4; the initial value of x is 5 then x=7.

Evaluating Expressions

Expressions without side effects:

Assume the value of the variables a=3,b=4,c=5

a * 4 + b /2 – c * b

Following are the rules:

1. Replace the variable by their values. This gives us the following expression: 3 * 4 + 4 /2 – 5 * 4
2. Evaluate the highest precedence operators, and replace them with the resulting value. In the above expression, the operators with the highest precedence are the multiply and divide. We therefore evaluates them first from the left and replace them with the resulting values. The expression is now (3 * 4) + (4/2)- (5*4) □ 12 + 2 – 20
3. Repeat step 2 until the result is obtained. Result=-6

Expressions with side effects:

Consider the expression --a * (3+b) /2 – c ++ * b

Assume the values of the variables a=3,b=4,c=5;

Following are the rules:

1. Calculate the value of the parenthesized expression (3+b)

   - -a * 7 /2 –c ++ *b.

2. Evaluate the postfix expression(c++). Remember that as a postfix expression, the value of c++ is same as the value of c; the increment takes place after the evaluation. The expression is now

   - -a * 7 /2 – 5 *b.

3. Evaluates the prefix expression (- -a) Remember that as a prefix expression, the value of - -a is the value after the side effect, which means that we first decrement a and then use its decremented value.

   2* 7 /2 – 5 *b

4. The multiply and division are now evaluated using their associativity rule, left to right. The expression is now

   14 /3 -5 * b 7-5 *4 7-20= -13

Note: In C, if a single variable is modified more than once in an expression, the result is undefined.

Type Conversions OR Mixed – Mode Expression

An expression that contains variables and constants of different data types is called as mixed – mode expression.

There are 2 types of data conversions that take place in mixed-mode expression.

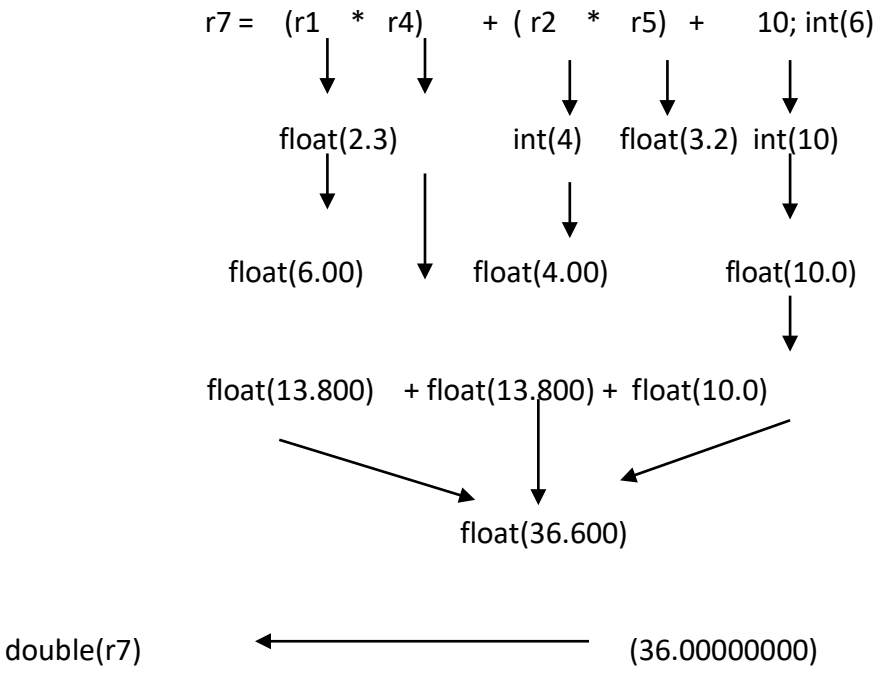
1. Implicit data conversion(Coercion)
2. Explicit data conversion(Type Casting)

1. **Implicit data conversion:** In this type the control will automatically convert data values of lower type are converted to the data types of higher types.

A. Implicit data conversion between integer and real values.

Example program:

```
main()
{
int r1=6,r2=4,r3;
float r4=2.3,r5=3.2,r6; double
r7;
```



printf("\n r7=%lf",r7);

Output: r7=36.00000000

Rule1: when a real value is assigned to an integer variable, the fractional part of the real value is truncated and only integer part is assigned to a variable.

Rule2: when an integer value say 10 is assigned to float variable, it is stored as 10.0000.

B. Implicit data conversion between character and integer value.

Example program

```
main()

{

char ch= 'a',op; int

x=65,y;


y = ch;
    char(a)

int(97)  ←

printf("\n ch=%c , y=%d", ch , y);//Output: ch=a, y=97 op

    =  x;
↑          ↓
char (a)  ← int(65)

printf("\n x=%d, op=%c", x, op);// Output: x=65,op=A
```

Rule 3: when a character is assigned to an integer variable, the ASCII value of the char is assigned to variable.

Rule 4: when a valid integer value (0 to 255) is assigned to char variable. Character equivalent of the integer value is assigned to a variable.

Note: Assigning an integer value into a character variable beyond its range. i.e. less than zero or greater than 255 leads to logical errors.

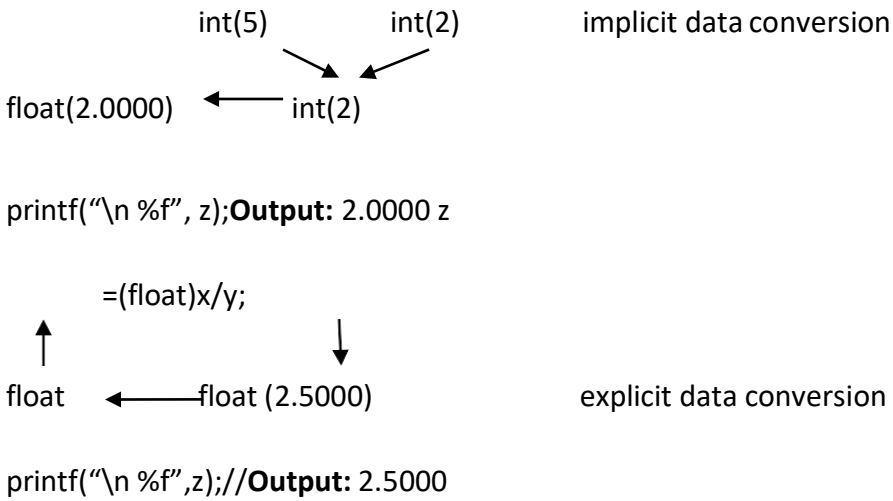
2. Explicit data conversion: In this type of data conversion between the output of an rvalue is forced to act as user specified data type. It refers to the process of changing an entity of one data type into another.

   Syntax: lvariable=(data type) rvalue;

   Typecasting

Example program for both explicit and implicit conversions

```
main()
        {
        int x=5, y=2;
        float z;
        z = x / y;
```



Statements:

A statement causes an action to be performed by the program. The C statement ends with a semicolon (;). The statement that starts with a '#' symbol is an "include statement" to include header files in the program. In C there are six types of statements. They are as follows.

1. Selection statements: These types of statements are executed depending on the condition. These include if and switch.
2. Iteration statements: These statements are executed repeatedly until a condition is satisfied. These include for, while and do-while.
3. Jump statements: It alters the flow of execution. These are break, continue, goto, and return.
4. Label statement: It is used to give the location of the target statement to which control must be transferred. These are case, default and label.
5. Expression statement: These statements composed of valid expression are called expression statements.
6. Block statement: A block consists of a set of statements enclosed within flower braces '{' and '}'. These are also called compound statements.

Control Structures

A program is nothing but the execution of sequence of one or more instructions. Some times it is desirable to alter the execution of sequence of statements in the program depending upon certain circumstances. This involves decision making through branching and looping**.** Control statements specify the order in which the various instructions in a program are to be executed by the computer. i.e. they determine the 'flow of control' in a program. Various control statements in C are:

- Selection or decision control statements
- Case control statements
- Repetition or Loop control statements

Selection control statements:- Selection control statements allow the computer to take a decision as to which statement is to be executed next. Different selection control statements are:

- **if** statement
- Conditional operator

❖ **if statement**: The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It can be achieved through different forms of if statement.

a) simple if b) if – else c) nested if-else d) else-if ladder

a) simple if statement: C uses the keyword "if" to execute a set of statements or one statements when the logical condition is true.

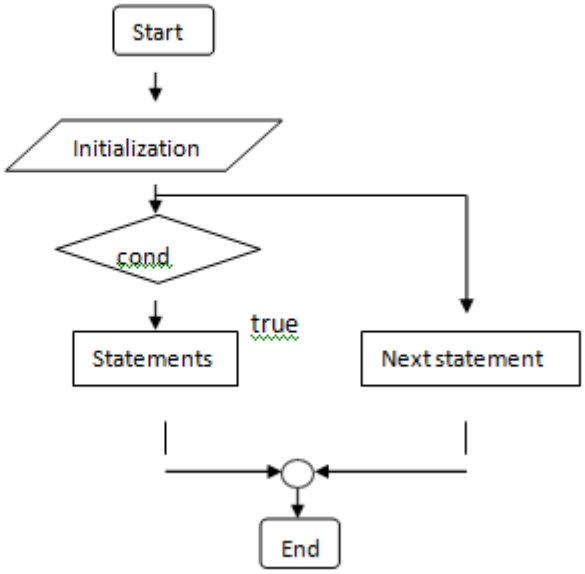
Syntax: if(condition or expression)

{ Statements-block;

}

next statement

Flow chart:



It allows the computer to evaluate the expression first and then depending on the value of expression, the control is transfers to the particular statement. If the expression is **true**(non-zero value) then the *statement-block* is executed and *next statement* is executed. If the expression is **false**(zero), directly next statement is executed without executing the *statement-block*. *Statement-block* may be one or more statements. If more than one statement, then keep all those statements in compound block({ })

Example:

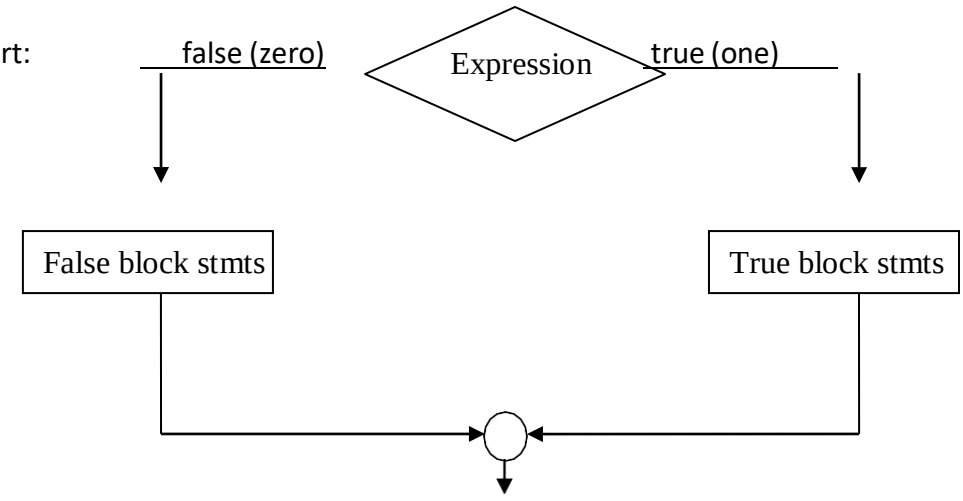
1. if(age > 60) printf("Old person");
2. if(marks >100)
   {
   printf(" Invalid marks");
   printf(" Check your marks once again ");
   }

b) if –else statement: It is an extension of simple if. It takes care of both the true as well as false conditions. It has two blocks. One is for if and it is executed when the condition is **true**, the other block is for else and it is executed when the condition is **false**. No multiple else statements are allowed with one if.

Syntax:

```
if(test condition)
{
        True block statements
}
else
{
        False block statements
}
Next statement
```

Flowchart:



If the condition is true then true block statement(s) is executed, otherwise if the condition is false, then false-block statement(s) is executed. The else cannot be used without if.

Example:

```
if(a%2 = = 0)
{
Printf(" even number");
}
else
{
Printf("odd number");
}
```

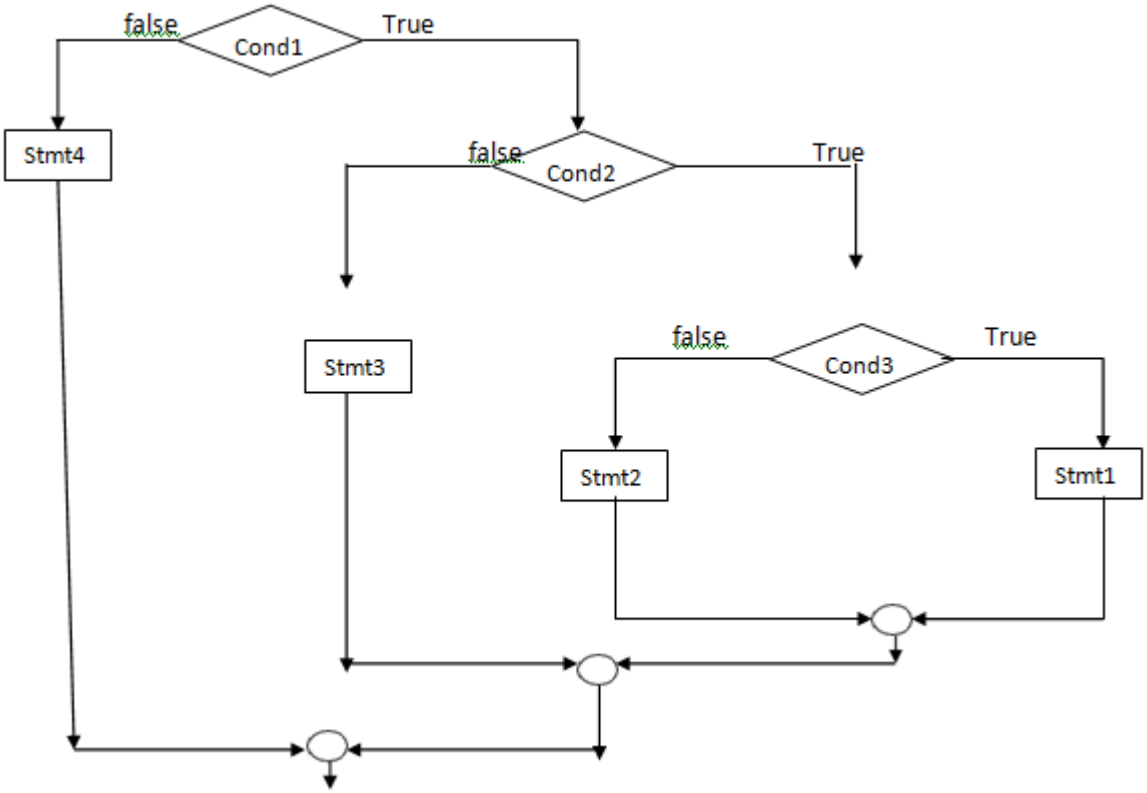
c) Nested if-else statement: When a series of decisions are involved, we have to use more than one if-else in nested form.

Syntax:

```
if( condition -1)
{
    if(condition -2)
    {
        ……
        if(condition-3)
        {
            Statement -1
        }
        else
        {
            Statement -2
```

```
                    }
            }
            else
            {
                    Statement-3

            }

}
else
{
        Statement-4

}
        next statement
```

Flow chart:



In this kind of statements, number of logical conditions is checked for executing various statements. If any condition is true then associated block will be executed, otherwise it skips and executes else block statements. We can add repetitively if statements in else block also.

Example:

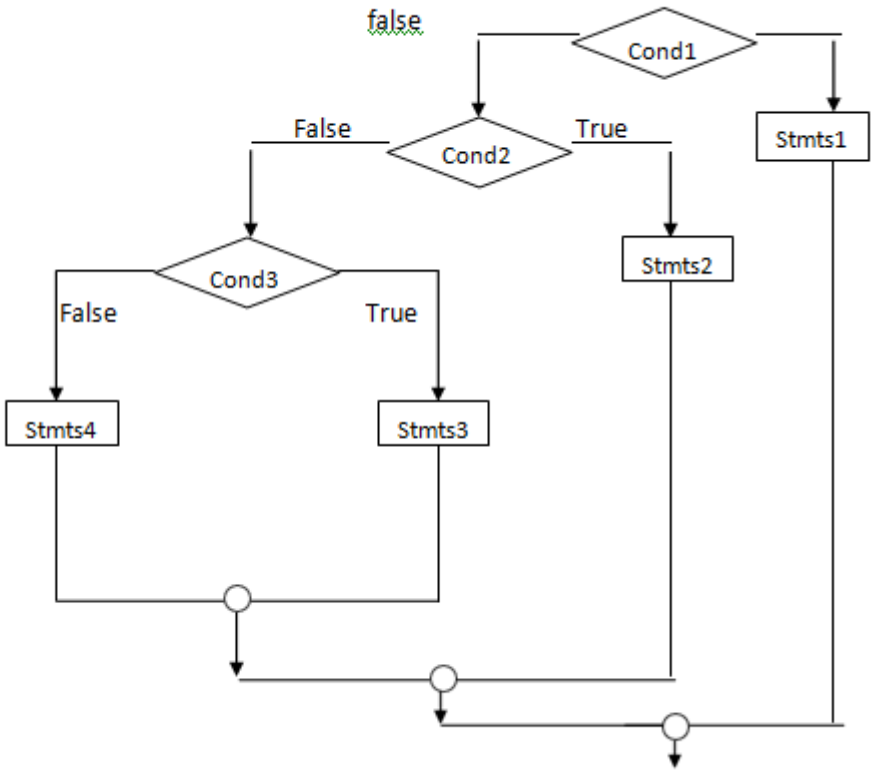
```
if(a>b)
{
if(a>c)
{
        printf(" ais big");
        else
        {
            printf("c is big");
        }
}
else
{
    if(c>b)
    {
        printf("c is big");
    }
    else
    {
        printf(" b is big');
    }
}
```

d) else-if ladder: The else-if ladder is used when multipath decisions are involved.

Syntax:

```
if(condition-1)
        Statement-1
else if(condition-2)
        Statement-2
else  if(condition-3)
          Statement-3
                ...
                ....
                ....
 else
          Statement-x
 next statement
```

Flow chart:



Example:

```
if(avg>=70)
   printf(" distinction");
else if(avg>=60)
        printf(" first class");
else if(avg>=50)
        printf("second class");
 else if(avg>=40)
         printf("third class");
 else
        printf("fail");
```

Case control statement: At times, the if condition may increases the complexity of the program when one of many alternatives is to be selected. C has built-in multiway decision statement known as ***switch-case*.** The switch statement requires only one argument variable or expression. It tests the value of a given variable against a list of case values and when a match is

found, a block of statements associated with that case is executed, if not such match, then default statement is executed.

Syntax: switch(variable or expression)

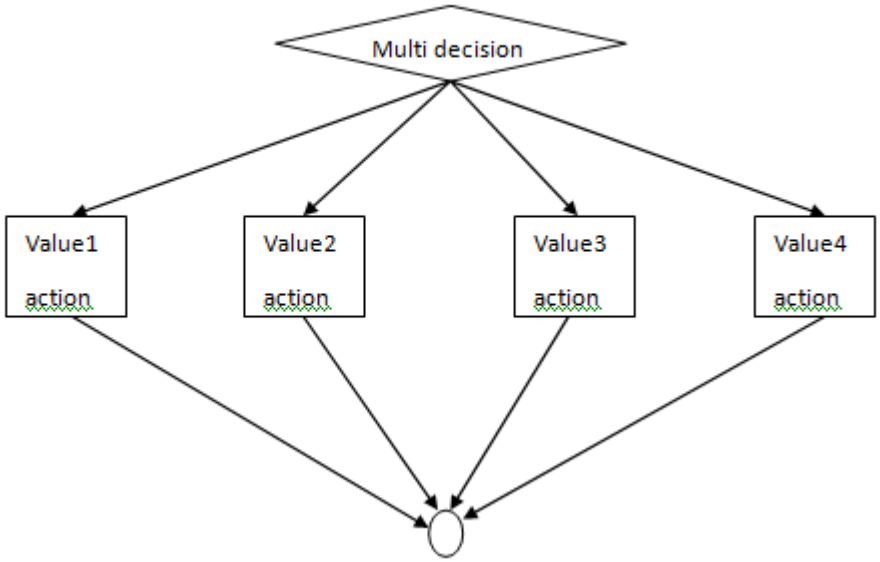
```
{

 case value-1:      block-1
                    break;
 case value-2:      block-2
                    break;
..

..

..

..

default:  default-block

}

next statement
```

The expression is an integer expression or character. Value-1, value-2 are either integer constants or character constants. These values should be unique with in a switch statement. **Case** labels end with colon (:).

The **break** statement signals the end of a particular case and causes an exit from the switch statement, transferring the control to the next statement following the switch.

The **default** is an optional case when present. It will be executed if the value of the expression doesn't match with any of the case values. If not present, no action takes place if all matches fail and control goes to the next statement.

Flow chart:



Example:

```
switch(op)
{
 case  '+':  c=a+b;  break;
case  '-':  c=a-b;  break;
case  '*':  c=a*b;  break;
case '/': c=a/b;break;
default: printf("wrong option");
}
```

Loop control statements

Many tasks are needed to be done with the help of a computer and they are repetitive in nature. Such type of actions can be easily done by using loop control statements.

Example: calculation of salary of employs of an organization for every month.

A loop is defined as a block of statements which are repeatedly executed for certain number of times to do a specific task.

Steps in loops:-

1. Loop variable: It is a variable used in loop to evaluate.
2. Initialization: It is the step in which starting value or final values are assigned to the loop variable.
3. Test-condition: It is to check the condition to terminate the loop. It is any relational expression with the help of logical operators.
4. Update statement: It is the numerical value added or subtracted to the loop variable in each round of the loop.

C language supports three types of loop control statements.

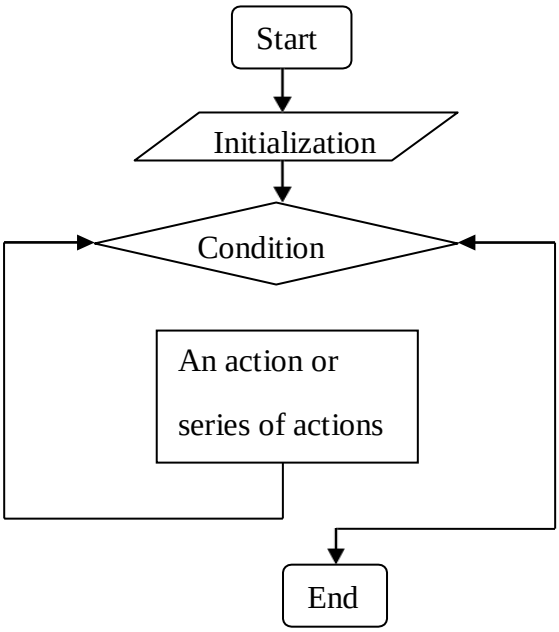
- **while**
- **do-while**
- **for**

- **while:** This is the simplest looping structure in C. the while is an entry-controlled loop statement.

Syntax:

```
initial statement

while(test condition)

{

        Statement(s) Update

        statement

}
```

Flow chart:



The test condition may be any expression, is evaluated and if it is true then the body of the loop is executed. The test condition is once again executed for updated values, and if it is true the body of the loop is executed once again. This process is repeated until the test condition is finally becomes false and control is transferred out of the loop to the next statement. The body of the loop may contain one or more statements. The braces are needed if the body of the loop contains more than one statement.

Example:

```
main()
{
    int i,sum;
    i = 1; sum=0;
   while (i<=10)
   {
          sum = sum + i ;
          i + +;
   }
   printf("sum=%d",sum);
}
```

Here the value, sum of first ten numbers is stored into the variable sum; **i** is called as loop variable.

The loop is repeated for ten times to do that process, each time by incrementing the value of i by one. Once the value of i becomes 11 then the test condition becomes false and the control is out of the loop.

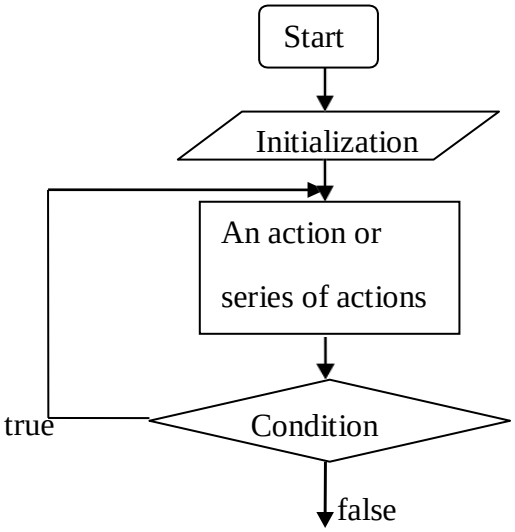
- **do-while**: On some occasions it might be necessary to execute the body of the loop before the test condition is performed. Such situations can be handled by the do-while statement. Do-while id exit controlled loop statement.

  Syntax:initial statement

```
do
{
        Statement(s)
        Update statement
} while(test – condition);
next statement
```

Flow chart:

(Post-test Loop)



The body of the loop is executed first, and then at the end of the loop the test condition is evaluated, if it is true then the statements are executed once again. The process of execution continues until the test condition finally becomes false and the control is transferred to the next statement

Example:

```
main()
{
    int i, sum;
    i =1;
    sum=0;
    do
    {
        sum = sum + i;
        i + +;
    } while( i<=10);
    printf("Sum=%d", sum);
}
```

❖ **for statement**: The for loop entry controlled loop that provides a more concise loop control structure.

Syntax:

```
for( initialization ; test-condition ; increment/decrement)
{
        Statement(s)
}
next statement
```

The for loop allows to specify three things about the loop in a single line.

i. Setting a loop counter variable to an initial value using assignment statement. Eg:

i=1 count=0;

ii. the test condition is a relational expression that determines the number of iterations desired or it determines when to exit from the loop. If the test condition is true, the body of the loop is executed, otherwise the loop is terminated and execution continues with the next statement after the loop.

Eg: i<=10

iii. After evaluating the last statement of the body the loop, the control is transferred to the increment/decrement statement of the loop. And the new value is again tested to see whether it satisfies the loop condition or not.

Eg: i++ ++i i+=2

- The body of the loop may contain one or more statements. In case there is more than one statement then braces
- The three sections of for loop must be separated by semicolons (;). Initialization and incr/decr parts may contain more than one statement must be separated by commas.
  Example: for(i=1, j=10 ; i<=10 ; i++, j--)
- The test-condition may have any compound relation and the testing need not be limited only to the loop variable.
  Example: for(i=1; i<20 && sum <100 ; i++)
- It is permissible to use expressions in the assignment statement of initialization and incr/decr sections.
  Example: for(k=(a+b)/2; k>0;k=k/2)
- The sections of for loop my absent depends on requirement in the program. But it leads to take some extra care about those sections.
  Example: for(; ;)

This statement leads to infinite loop or never–ending process.

Nesting of loops: The way if statement can be nested, similarly whiles and for can also be nested.

```
Example:    for(i=1;i<=3;i++)        //outer for loop
            {
            for(j=1;j<=2;j++)        // inner for loop
            {
                    Printf("\t %d   %d", i,j);
            }
            Printf("\n");
      }
```

Here in this example nested for loop is used, and the total process is executed for 6(3 * 2)

times. And the output of this example will be:

1 1 1 2

2 1 2 2

3 1 3 2

The way for loops have been nested here, similarly while and do-while can also be nested. Not only this, a for loop can occur within a while loop, or a while within a for.

Example: Comparison of three loops. Finding the sum of first 10 numbers by using all loop statements.

Program to write the sum of first 10 numbers by using *while* loop.

```
#include<stdio.h> int
main()
{
        int i,sum;
        sum=0; i=1;
        while(i<=10)
        {
                sum=sum+i;
                i++;
        }
        printf("Sum of first 10 numbers = %d", sum);
        return 0;
}
```

Program to write the sum of first 10 numbers by using *do-while* loop.

```
#include<stdio.h> int
main()
{
        int i,sum;
        sum=0;
        i=1;
```

```
        do
        {
                sum=sum+i;
                i++;
        }while(i<=10);
        printf("Sum of first 10 numbers = %d", sum);
        return 0;
}
```

Program to write the sum of first 10 numbers by using *for* loop.

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

Jumps in loops: We often come across some situations where we want to make a jump from one statement to other statement, jump out of a loop or to jump to next iteration of the loop instantly,. This can be accomplished by the statements like :

- **break**
- **continue**
- **goto**.

break statement:

- When we want to jump out of a loop instantly without waiting to get back to the condition test, then the keyword *break* allows us to do this.
- The break statement provides an early exit from the loop.
- A break is usually associated with an if.

Example:

```
for(i=1;i<=3;i++)
{
    for(j=1;j<=5;j++)
    {
        if(j = = 3)
            break;
        else
            printf(" %d %d", i, j);
    }
    printf("\n");

}
```

Output:

```
1  1   1  2

2  1   2  2

3  1   3  2
```

In this example when j value equals 5, break takes the control outside the inner for loop only, since it is placed inside the inner loop.

The continue statement:

- When we want to take the control to the beginning of the loop by passing the statements inside the loop which are not yet been executed, then the keyword *continue* allows us to do this.
- It causes the next iteration of the loop to begin and it applies only to loops.
- A continue is usually associated with an if.

Example:

```
for(i=1;i<=2;i++)
{
    for(j=1;j<=2;j++)
    {
        if(i = = j)
            continue;
        else
            printf(" \n%d %d", i, j);
    }
}
```

Output :
1 2
2 1

In this example when i and j values are equal, *continue* takes the control to the inner for loop by ignoring rest of the statements in the inner for loop.

The goto statement:

- The goto statement alerts the normal sequence of program execution and control transfers to some other part of the program.
- The goto statement leads the control to go to the particular part of the program by indicating in its label name.

Syntax: goto *label;*

Where label is a unique identifier used to label the target statement to which control will be transferred. The general format of the label statement is:

label : statement;

Example:

```
void main( )
{
        int x;
        read: scanf("%d",&x);
        if(x<0)
            goto read;
         else
            printf("x=%d",x);
}
```

Here read is label, whenever x value entered as negative(x<0) then the control goes to read label, and once again scanf( ) statement executed to read next value.

Command Line Arguments: Arguments passed to the main function are known as Command Line Arguments. Command line arguments are given after the name of the program during the execution of the program in a command-line shell. To pass command line arguments, main() is defined with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

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

- argc **(arg**ument **c**ount) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- argv(**arg**ument **v**ector) is array of character pointers listing all the arguments.
- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- argv[0] is the name of the program , After that till argv[argc-1] every element is a command line argument.

Example:

```
int main(int argc, char* argv[ ])
{
     printf( "You have entered  %d arguments ", argc );
     for (int i = 0; i < argc; ++i)
    printf("%s", argv[i] );
    return 0;
}
```

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[0] holds the name of the program.
5. argv[1] points to the first command line argument and argv[argc-1] points last argument.

UNIT-II

Topics Covered:

Pointers – Pointer variable, pointer declaration, Initialization of pointer, Accessing variables through pointers, Pointer Arithmetic, pointers to pointers, void pointers

Arrays – Definition, declaration of array, Initialization, storing values in array, Two dimensional arrays, Multi-dimensional arrays. Arrays and Pointers, Array of pointers

Strings – Declaration and Initialization, String Input / Output functions, String manipulation functions, strings and pointers, Arrays of strings

Pointer: A Pointer is a variable which can store the address of another variable

Syntax for declaring a Pointer

datatype *var_name

datatype is the pointer's base type; it must be a valid C data type and **var-name** isthe name of the pointer variable

Example: int *p; - Pointer to an integer float
*q; - Pointer to a float
char *r; - Pointer to a char

a. We define a pointer variable
b. assign the address of a variable to a pointer
c. access the value at the address available in the pointer variable

Example Program for declaring and using a Pointer

```
#include<stdio.h> int
main( )
{
int i = 10, *p; p = &i;
printf("Value of i through pointer p : %d", *p); return 0;
}
```

Benefits of using pointers are

a. Pointers can be used to return multiple values from a function via function arguments.
b. Pointers allow arguments to be passed by reference
c. Pointers allow passing arrays and strings as arguments to functions more efficiently
d. Pointers allow C to support dynamic memory management.

Null Pointer: A pointer that is assigned NULL is called a **null** pointer.

Example:int *p = NULL

Pointer to Pointer: A variable that holds the address of another Pointer variable

Syntax: datatype **var_name;
Example: int **q;

Program showing the use of Pointer to Pointer

```
#include<stdio.h> int
main( )
{
int a=10, *p, **q; p=&a;
    q=&p;
printf("Value of a : %d", a);
printf("Value of a through pointer p : %d", *p); printf("Value of a
through pointer to pointer q : %d", **q); return 0;
}
```

Void Pointer: A variable that can hold the address of a variable of any type

Syntax: void *var_name;
Example: void *p;
Program showing the use of Void Pointer

```
#include<stdio.h> int
main( )
{
int a = 10;
float b = 10.5; char c =
'A'; void *p;
    p=&a;
printf("Value of a through pointer : %d", *((int*)p)); p=&b;
printf("Value of b through pointer : %f", *((float*)p)); p=&c;
printf("Value of c through pointer : %c", *((char*)p));
return 0;
}
```

Arithmetic operations that can be performed on pointers

i. Like normal variables, pointer variables can be used in expressions.

Example: x= (*p1) + (*p2);

ii. C language allows us to add integers to pointers and to subtract integers from pointers

Example: If p1, p2 are two pointer variables then operations such as p1+4, p2 - 2, p1 - p2 can be performed

iii. Pointers can also be compared using relational operators.

Example: p1>p2, p1==p2, p1! =p2 are valid operations

iv. Pointer constants should not be used in division or multiplication. Also, two pointers cannot be added.

Example: p1/p2, p1*p2, p1/3, p1+p2 are **invalid** operations.

v. Pointer increments and scale factor

when a pointer is incremented, its values is increased by length of data type that it points to. This length is called scale factor.

Example: If the address of p1 is 1002. After using p1=p1+1, the value becomes 1004 but not 1003.

Arrays:

C supports a *derived data type* known as *array* that can be used to handle large amounts of data (multiple values) at a time.

- An ***array*** is collection of elements of the *same data type*, which share a common name. Array elements are always stored in *contiguous* memory locations.
- Each element in the group is referred by its position called *index*.
- The first element in the array is *numbered 0*, so the last element is one less than the size of the array.
- Before using an array its type and dimension must be declared, so that the compiler will know what kind of an array and how large an array.

Each array element is referred by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets. The value of each subscript must be expressed as a *non-*

negative integer, variable or expression. The number of subscript determines the dimensionality of the array. We can use arrays to represent not only list of values but also tables of data in two or more dimensions. We have different types of arrays based on its dimensional.

- One-dimensional arrays / Single-dimensional arrays
- Two-dimensional arrays / Double-dimensional arrays
- Multi-dimensional arrays

One – Dimensional arrays:

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

Syntax: data-type array-name[size];

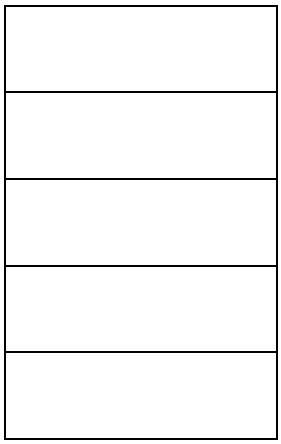
The data-type specifies the type of elements such as int, float or char. And the size indicates the maximum number of elements that can be stored in that array

Eg: int a[5] ; here a is an array containing 5 integer elements.

char name[20]; here name is an array containing 20 characters.

Note: C language character *strings are* as simple as *array of characters*. And every string should be terminated by *null character* ***('\0')***.

The computer internally reserves five storage locations for array **a** in the above example as follows.



These elements may be used in programs just like any other C variables. Eg:

```
x = a [3] + a [2];

Y= a [4] * 2;
```

Note: C performs no bounds (array size) checking and therefore care should be exercised to ensure that the array indices are within the declared limits.

Initialization of one-dimensional arrays: After an array is declared its elements must be initialized. Otherwise they will contain garbage values. An array can be initialized at either of the following stages.

- At compile time
- At run time

Compile time initialization:

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The values in the list must and should be separated by comma.

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

Eg:

- ✓ int a [6] = { 3, 5, 23, 28, 6, 11 }; will initialize the given values to array a.
- ✓ float avg [5] = { 78.45, 65.92, 59.1 } will assign the first three elements to 78.45, 65.92 and 59.1 and the remaining two elements to zero.
- ✓ int counter [ ] = { 1, 1, 1, 1 }; here the size is optional, it may be omitted. In such cases the compiler allocates enough space for all initialized elements.
- ✓ char name[ ] = { 'S', 'I', 'M', 'H', 'A', '\0' }; here declares the name to be an array of characters, and is initialized by the string "SIMHA" ending with the null character. And we can do the same thing in another way also. i.e. we can assign the entire string at a time, not an element by element as follows:

  char name [ ] = "SIMHA"

- ✓ intnum [3] = { 10,20,30,40,50 }; will not work. It is illegal in C. i.e. if we have more initializes

than the declared size, the compiler will produce an error.

Run time initialization: An array can be explicitly initialized at run time. At the time of execution array values can be given.

Examples:

- ✓ int a[10] ; for (i=0;
  i<10; i++)

  scanf("%d",&a [i]);

- ✓ float x [15]; for(i=0;
  i<15; i++)

  scanf("%f", &x [i]);

- ✓ char name [10];
  for(i=0;i<5;i++)

  scanf("%c", &name [i]); or scanf(" %s", name); or

  char name[10];

  gets(name);

//Example for One dimensional array – Sum of the elements of an array
// Initialize the array elements at compile time #include<stdio.h>

```
int main()
{
        int     i,  sum=0;
        int     a [ 10 ] = { 12, 3, 5, 43, 21, 1, 6, 8, 9, 11 };
        // Array a is initialized at compile time
        /* finding the sum of array a */

        for(i=0; i<10; i++) sum =
        sum + a[ i ];
```

```
        printf("\n Sum of elements of array a is %d", sum); return 0;
}
```

```
//Example for One dimensional array – Sum of the elements of an array
// Initialize the array elements at run time #include<stdio.h>
int main()
{
        int     a[ 10], i, n, sum=0;
        printf ("Enter no.of elements :"); scanf
        ("%d", &n);
        printf("Enter %d elements :",n); for(i=0;
        i<n; i++)
        scanf("%d",&a [ i ]);                   // run time initialization
        /* finding the sum of array a */ for(i=0;
        i<n; i++)
        sum = sum + a[ i ];
        printf("\n Sum of elements of array a is %d", sum); return 0;
}
```

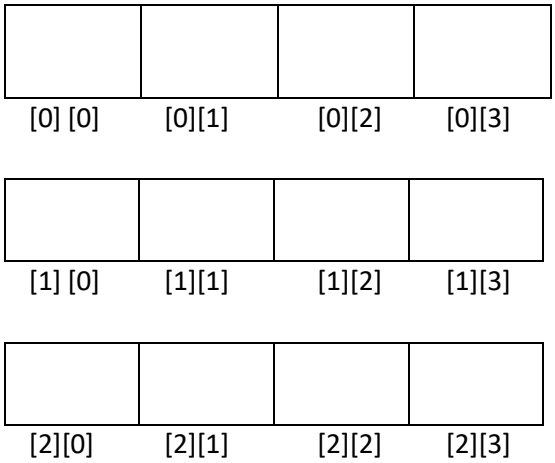
Two - Dimensional Arrays: C allows us to define the data in the form of table of items by using two-dimensional arrays.

Syntax: data-type array-name [row size] [column size];

Eg: int a [3] [4] ; Here a is two dimensional array with row size 3 and column size 4., and the total elements we can store in this array a is 12 (i.e. 3 * 4).

Each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column with in that row. Two dimensional arrays are stored in memory as follows:

Examples: Representation of two dimensional array in memory



Initializing Two Dimensional Arrays: Like on-dimensional arrays, two-dimensional arrays can also be initialized at compile time and runtime.

Compile time initialization:

Two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. Until and unless we initialize the array explicitly it is having some garbage values initially.

Eg:- int a [2] [3] = {3, 5, 7, 12, 22, 32};
Here a[0][0] = 3 a[0][1] = 5 a[0][2] = 7

And a[1][0] = 12 a[1][1] = 22 a[1][2] = 32

The initialization can also be done in row by row as follows. int a

[2] [3] = { { 3, 5, 7 } , {12, 22, 32 } };

We can also initialize a two dimensional array in the form of a matrix as follows: int

a[2] [3] = {

{ 3, 5, 7 },

{12, 22, 32 }

};

When the array is completely initialized with all values, explicitly we need not specify the size of the first dimension (row size). i.e

```
Int     a [ ] [ 3] = {

                        { 3, 5, 7 },

                        {12, 22, 32 }

                        };
```

if the values are missing in an initialize, they are automatically set to zero. i.e. int

```
a [2] [3] = { { 1, 5, },
              {12}
            };
```

So here the first two elements of the first row are initialized by 1 and 5 respectively, the first element of second row to 12, and **all other** elements to **zero**.

Run time Initialization: An array can be explicitly initialized at run time. Eg:

```
int     a [2] [3] ;
Int     i, j;

for ( i=0 ;  i < 2 ;  i++)          // To represent row index ( i= 0,1 )
{
        for ( j =0 ; j < 3 ; j ++ ) // To represent column index ( j = 0,1,2)
           scanf( " %d", & a[ i ] [ j ]);
}
```

Eg: Program to print the multiplication table for first 10 numbers up to the 10 multiples.

```
#include<stdio.h>
    #define         ROW         10
    #define         COLUMN      10
    int main()
    {
      int     i,j,r,c,table[ROW][COLUMN];
```

```
        printf("\n              MULTIPLICATION TABLE \n\n");
        printf("X |");

        for(j=1;j<=COLUMN; j++)
            printf("%5d",j);
  printf("\n_____________________________________________________\n\n");
        for(i=0;i<ROW;i++)
        {
r=i+1; printf("%d | ",r);
for(j=1;j<=COLUMN;j++)
{
 c=j;
table [i][j]= r*c;
 printf("%5d",table[i][j]);

}
printf("\n\n");
}
return 0;
}
```

In the above program the size of the array table is mentioned as ROW and COLUMN (i.e. **table[ROW] [COLUMN])** where these **symbolic constants** having the values 10 and 10. It is always good practice to take the size of the array in terms of symbolic constants instead of directly taking the values.

Multi-Dimensional arrays:

C allows arrays of three or more dimensions. The compiler determines the exact limit.

Syntax: data-type array-name $[S_1][S_2][S_3]\ldots\ldots[S_n]$; Where

S_i is the size of the i^{th} dimension.

Eg: int table [3] [4] [2]; Here table is the three dimensional array containing 24 integer elements (i.e. 3 * 4 * 2 = 24).

float a [4] [2] [5] [3]a is a multidimensional array containing 120 elements.

NOTE: A three dimensional array can be represented as a series of two-dimensional arrays, and a two-dimensional array can be represented as a series of one dimensional arrays.

```
/* PROGRAM TO DECLARE AND INITIALIZE A MULTI-DIMENSIONAL ARRAY AT COMPILE TIME AND PRINT THE SAME
*/
#include<stdio.h>
 void main()
{
int a[3][2][3]={ { {1,1,1},
{1,1,1}
},
{ {2,2,2},
{2,2,2}
},
{ {3,3,3},
{3,3,3}
}
};
inti,j,k;
printf("\n A three dimensional array is :\n");
 for(i=0;i<3;i++)
{
for(j=0;j<2;j++)
{
for(k=0;k<3;k++)
{
printf(" %d",a[i][j][k]);
}
printf("\n");
}
printf("\n\n");
}

return 0;
}
/* PROGRAM TO DECLARE AND INITIALIZE A MULTI-DIMENSIONAL ARRAY AT RUNTIME AND PRINT THE SAME */
#include<stdio.h> void main()
{

int a[5][5][5];

inti,j,k,x,y,z;

printf("\n Enter the size of a 3D array ");
scanf("%d%d%d",&x,&y,&z);
printf("\n Enter values into a 3D array of size %dx%dx%d \n",x,y,z);

for(i=0;i<x;i++)
{
for(j=0;j<y;j++)
{
```

```
for(k=0;k<z;k++)
{
scanf("%d",&a[i][j][k]);
}
}
}
printf("\n A three dimensional array is :\n");
for(i=0;i<x;i++)
{
for(j=0;j<y;j++)
{
for(k=0;k<z;k++)
{
printf(" %d",a[i][j][k]);
}
printf("\n");
}
printf("\n\n");
}
return 0;
}
```

Arrays and Pointers

Elements of an array can be accessed through a pointer also. We can declare a pointer variable and store the address of the first element of the array in the pointer variable. All the elements of the array can be accessed through the pointer.

Example: int arr[10] = {1,2,3,4,5};

int *ptr = arr;

In the above example, arr is an integer array, ptr is an integer pointer in which address of the first element of the array is stored

Program to access the elements of an array through pointer

```
#include<stdio.h>

int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };        // integer array
    int *ptr = arr;                        // pointer ptr storing the address of the first element of arr

    for(i=0;i<5;i++) printf("%d\n",
      *ptr);
   return 0;
}
```

Array of Pointers(Pointer arrays):

An **array of pointers** is an indexed set of variables in which the variables are pointers (a reference to a location in memory).

Syntax: datatype *arr_name[size];

Example: int *ptr[5];

Here ptr is an array name whose elements are integer pointers

Program for Array of Pointers

```
#include<stdio.h> int main( )
{

int a=10, b=20, c=30; int *ptr[3];

p[0]=&a;
p[1]=&b;
p[2]=&c;

printf("Value of a : %d",*p[0]);
printf("Value of b : %d",*p[1]);
printf("Value of c : %d",*p[2]);
return 0;
}
```

Strings

A string is an array of characters. (Or) A string is a one-dimensional array of characters terminated by a null character ('\0').

Eg: "INDIA" "WELCOME"

Each character of the string occupies one-byte of memory. The characters of the string are stored in contiguous memory locations.

Declaration and Initialization of Strings Syntax:

char string-name [size]; Eg:

char city [20];

Character arrays (strings) may be initializing when they are declared (compile time initialization).

Eg:

- ✓ We can initialize a character array as element by element and last character should be the null character.

```
char city [10] = { 'H', 'Y', 'D', 'E', 'R', 'A', 'B', 'A', 'D', '\0' } ;
```

- ✓ We can initialize a character array as total string at a time.

```
char city [10] = "HYDERABAD";
```

- ✓ We can initialize a character array with out specifying the size, the compiler automatically determines the size.

```
char city [ ] = "HYDERABAD";
```

The memory representation of this character array is as follows:

0	1	2	3	4	5	6	7	8	9
H	**Y**	**D**	**E**	**R**	**A**	**B**	**A**	**D**	**\0**
3010	3011	3012	3013	3014	3015	3016	3017	3018	3019

Reading Strings:-

- ✓ The input function scanf ( ) can be used with %s format specification to read a string. Eg:

```
char name[30];
scanf("%s", name);
```

- ✓ The scanf() function to read multi word strings is doesn't consider white spaces. It terminates the input string at first occurrence of white space. So multi word strings can't read by using scanf() function.

Eg:

```
char city[30]; scanf("%s",
city);
```

If the input string is "NEW DELHI", then the variable city can store only "NEW", it ignores the rest because white space is occurred after NEW.

✓ To read multi word strings, we can use **gets( )** function or **getchar( )** function repeatedly. Eg:

```
char    city[20];
        gets(city);
```

if the input string is "NEW DELHI", then name contains total string "NEW DELHI".

(Or)

```
int     i=0;
char city[20], ch; do
{
        ch=getchar(); name[ i
        ]=ch; i++;
} while(ch != ' \n');
```

✓ C language doesn't allow operators that work on stings directly. I.e. assignment, arithmetic, equality operator etc are doesn't work with strings.

Eg: char s1, s2;

s1 = "WELCOME" □ invalid

s1==s2 □ invalid

s1+s2 □ invalid

Writing of strings to screen

✓ The printf() function with %s specification is used to print strings to the screen. Eg:

```
printf("%s",city);
```

✓ We can also specify the precision with which the array is displayed. Eg:

```
char    name[ ] =" RADHA KRISHNA";
```

printf(" %s", name);	RADHA KRISHNA
printf(" %**14**s", name);	RADHA KRISHNA
printf(" % **5**s", name);	RADHA KRISHNA
printf(" %**14.5**s", name);	RADHA
printf(" %-**14.5**s", name);	RADHA

printf(" %**.3**s", name); RAD

We can also print strings by using puts () function and putchar () function repeatedly

Eg: puts(name)

OR

```
for(i=0;name[i]!='\0';i++)
{
    putchar(name[i]);
}
```

PROGRAM TO READ A STRING AND PRINT IT

```
#include<stdio.h> int main()
{
char ch, city[20]; int i;
printf("Enter a city name ");
 i=0;
do
{
ch=getchar();
 city[i]=ch;
i++;
}while(ch!='\n');
 i=i-1;
 city[i]='\0';
printf("The city is :");
 puts(city);
return 0;
}
```

String handling functions

Operators can't work with strings directly. So to manipulate strings we have a large number of string handling functions in C standard library and the responsible header file is *"string.h"*. Some of those functions are:

strcpy ( ), strcmp ( ), strcat ( ), strlen ( ), strrev ( ) etc.

strcpy( ): It is to copy one string into another, and it returns the resultant string.

Syntax: strcpy(string1, string2);

String2 is copied into string1.

Eg: char city1[10] = “HYDERABAD”;

char city2[12] = “BANGLORE”;

strcpy (city1, city2);

Here city2 is copied into city1. So city1= “BANGLORE” and city2=”BANGLORE”. The size of the array city1 should be large enough to receive the contents of city2.

strcmp( ): It is to compare two strings to check their equality. If they are equal it returns zero, otherwise it returns the numeric difference between the first non matching characters in the strings. (i.e. +ve if first one is greater, -ve if first one is lesser).

Syntax: strcmp (string1, string2);

Eg: char city1[10] = “HYDERABAD”;

char city2[12] = “BANGLORE”;

strcpy (city1, city2);

Here city1 and city2 are compared, and returns the numeric difference between ASCII value of ‘H’ and ASCII value of ‘B’ as they are not equal.

Strcmp(“RAM”,ROM”); It returns some –ve value.

Strcmp(“RAM”,RAM”): It returns 0 as they are equal.

strcat ( ): This function is used to join two strings together, and it returns the resultant string.

Syntax: strcat(string1, string2);

String1 is appended with string2 by removing the null character of string1 and string2 remains unchanged. The resultant string is stored in string1.

Eg: char city1[10] = “HELLO”;

char city2[12] = “WORLD”;

strcat (city1, city2);

Here city2 is appended to city1. So city1= “HELLOWORLD” and city2=”WORLD”.

strlen( ): It is to find out the length of the given string and it returns an integer value, that is the number of characters in the given string. It takes only one parameter

Syntax: strlen(string1); It gives the length of string1.

Eg: char city[10] = “HYDERABAD”; strlen (city);

 it gives the value 9.

strrev ( ): This function is to find out the reverse of a given string.

Syntax: strrev (string);

Eg: char city[10] = “HYDERABAD”;

strrev (city); it give the string “DABAREDYH”

Strings and Pointers

A pointer can be used to access the individual elements of a String.

For Example char

str[10];

declares a string str. The variable name of the string str holds the address of the first element of the array i.e., it points at the starting memory address.

So, we can create a character pointer ptr and store the address of the string str variable in it. This way, ptr will point at the string str.

```
char *ptr= str;
```

In the following code we are assigning the address of the string str to the pointer ptr. #include<stdio.h>

```
int main()
  {
     char str[6] = "Hello";    // string variable
    char *ptr = str;                // pointer variable points to the first address of the string
     // print the contents of the string using pointer
     while(*ptr!='\0')
```

```
        {
            printf("%c", *ptr);
            ptr++; // increment ptr so that it points to the next element of the string
         }
        return 0;
}
```

Array of Strings

We have array of integers, array of floating point numbers, etc..similarly we have array of strings also. Collection of strings is represented using array of strings.

Declaration:

Char arr[row][col]; where,

arr - name of the array

row - represents number of strings col -

represents size of each string

Initialization:-

chararr[row][col] = { list of strings };

Example:-

char city[5][10] = { "DELHI", "CHENNAI", "BANGALORE", "HYDERABAD",
"MUMBAI" };

D	E	L	H	I	\0				
C	H	E	N	N	A	I	\0		
B	A	N	G	A	L	O	R	E	\0
H	Y	D	E	R	A	B	A	D	\0
M	U	M	B	A	I	\0			

In the above storage representation memory is wasted due to the fixed length for all strings

```
#include<stdio.h>

int main()
{
   int i;

            char ch_arr[3][10] = {
                        "spike",
                "tom", "jerry"
              };

   for(i = 0; i < 3; i++)
   {
      printf("string = %s \n", ch_arr[i]);
   }
     return 0;
}
```

UNIT-III

Topics Covered:

Designing Structured Programs using Functions - Types of Functions- user defined functions, Standard Functions, Categories of functions, Parameter Passing techniques, Scope
– Local Vs Global, Storage classes, Recursive functions.
Passing arrays as parameters to functions, Pointers to functions, Dynamic Memory allocation

DESIGNING STRUCTURED PROGRAMMMES

In the preceding chapters, we used only the three non-derived types, void, integral, and floating – point. These are used to solve only the limited amount of problems. There are six derived data types in C: Array, Functions, Pointer, Structure, Union, and Enumerated. Breaking a complex problem into smaller parts (understandable parts). We call each of these parts of a program a module and the process of subdividing a problem into manageable parts **top-down design.** In the principle of top-down design and structured programming, a program is subjected to factoring, that is the program is divided into main module and its related modules. Each module is in turn divided into sub modules until the resulting modules cannot be further subdivided. A **C program** finds itself in a similar fashion as in top-down structured designing. It cannot handle all the tasks by itself. Instead it requests other program-like entities – called **functions**.

Function: Definition

A **function** is a self-contained block of one or more statements that perform a particular task. Every C program can be thought of as a collection of these functions, and one of which must be main ( ) function.

The execution of the program always starts and ends with main function and main function can call other functions. A **called function** can receive control from a **calling function**, when the called function completes its execution it returns the control back to the calling function. The communication between calling function and called function is by **passing parameters** at the time of calling.

Advantages of functions in C

- It facilitates top-down modular programming; here the problem can be factored into understandable and manageable steps.
- Reusing of the code: The length of the source program can be reduced by using functions at appropriate places. i.e writing functions avoids rewriting of same code over and over.
- Using functions it becomes easier to write programs and keep track of what they are doing.
- The functions are much easier to understand and test.
- C comes with rich and valuable set of library functions that makes programmer's work easier.

C functions can be classified into two categories.

- Library functions
- User-defined functions

Library functions: The library functions are predefined set of functions. The user can only use these functions but can't change or modify them. These functions are already defined, tested and debugged in C standard library. When the user needs to use library functions, then a call to these functions is required.

Eg: The basic input and output functions like printf( ) , scanf( ) are defined in the C library header file stdio.h.

sqrt( ), sin( ), cos( ), tan( ), pow( ), getchar( ), exit( ), toupper( ), strcmp( ), strlen( ) etc., all these are some C library functions.

User-defined functions: These are defined by the user according to the requirement in the application. The user can modify and can create any number of functions based on requirement. The user can certainly understand the internal working of the function.

The difference between library functions and user-defined functions

- The library functions are not required to written by the user, where as user-defined functions have to develop by the user at the time of writing of application.
- The user can't modify the meaning of the library functions. But user can modify the user-defined functions.
- The library functions are already tested and debugged, but the user-defined functions need to be test and debug.

The structure of user-defined function:

Syntax: **return-type function-name (arguments list)**

argument declaration

{ Local variable declarations

Executable statement

……

……

……

return (expression or value)

}

- In function structure all parts not essential, it is based on the requirement of the program and some sections or parts may be absent.
- Return-type is any data type
- All functions by default return int type data.
- Function-name can be any name like variables which follows naming conventions of identifiers.
- The arguments list and its associated declaration are optional. These are any valid variable name separated by commas. There may be any number of variables, any type of variables with respect to the function calling statement and these variables receive values from calling function.
- The declaration of local variables is required only when any local variables are used in the function.
- A function can have any number of executable statements.
- The ***return*** is a keyword which is followed by some expression or value. The return statement returns a value to the calling function and is optional. When there is no return statement, then no value is being returned to the calling function. We can **return only one value at a time**.

Eg:

```
int add( int x, int  y)
{
int z;
z = x + y;
return z;
}
```

(**or**)

```
int add(x,y)
int x, y;
{
  int z;
  z = x + y;
  return z;
}
```

(**or**)

```
int add( int x, int y)
{
return ( x + y );
 }
```

- Every user-defined function can have its declaration (also called as ***function prototyping***) and definition. Some other function can call this function.
- When function calling statement is invoked, the control is transferred to the function definition. It is then executed and the control is transferred back to the calling function to execute next statement.
- Any function can call any other function and can call itself (recursion).
- A function can call any number of times.
- C permits nesting of functions. Main can call function-1, which calls function-2, which calls function-3 and so on.

Actual arguments and formal arguments:

- The arguments (parameters) of function **calling** statement are **actual** arguments; the arguments of **called** function are **formal** arguments.
- The actual and formal arguments should match in number, type and order.
- The formal arguments receive values from actual arguments when a function is calling, and by this a communication between calling function and called function is made.
- When a function call is made, only a copy of the values of actual arguments is passed into the called function's formal arguments.
- The values of the actual arguments must be assigned before the function call is made.
- The formal arguments must be valid variable names, but the actual arguments may be variable names, expressions, or constants.

Eg: //program for addition of two numbers #include

```
<stdio.h>

int main()

{

    int a, b, c;

    int add(int, int);          // function declaration or prototyping
    printf("Enter two values ");

    scanf("%d%d", &a, &b);

    c = add(a,b);    //function calling statement printf("

    addition of a, b is %d", c);

    return 0;

}

int  add( int x, int y)              // add function definition

{

    int z;

    z = x + y; return z;

}
```

In the above program:

- a, b are actual arguments
- x, y are formal arguments
- main( ) is calling function.
- add( ) is called function

The values of a, b are passed into x, y respectively when function is invoked and a, b values remains same.

The return statement:-

- Functions uses return statement to return a value to the calling function. And exit from the called function is done by the return statement.
- The return statement returns only one value at a time. There may be more than one return statement in a function, but it returns only one value.
- The value returned by the called function is collected by the calling function.
- Return statement with out any value will return value one. Eg:

  return; this returns 1 to the calling function

- Absence of return statement indicates that no value is returned, such functions return type is void (i.e returning nothing).

Eg; return; return (9*3); return (a + b); return c;

Function prototype:

- Function prototyping is nothing but function declaration; it consists of function's return type, name, and arguments list with type.
- When a function is defined, the function header in the definition must be same like its prototype declaration.
- The function prototype statement is terminated by semicolon.
- A prototype statement helps the compiler to check the return type and arguments type of the function.
- The prototype of library functions is given in the respective header files, those we include by using #include preprocessor directive.

Eg: int add(int, int); void display(char c); float average(float, float, float);

Category of functions: A function depending on whether the arguments are present or not, and a value is returned or not may belong to one of the following categories.

1. Functions *without* **arguments** and *without* **return values**.
2. Functions *with* **arguments** and *without* **return values**.
3. Functions *without* **arguments** and *with* **return values**.
4. Functions *with* **arguments** and *with* **return values**.

1. Functions *without* **arguments** and *without* **return values:**

- When a function has no arguments, it does not receive any data from calling function.
- When a function does not return any value, the calling function doesn't receive any data from called function.
- There is no data transfer in between the calling function and called function.
- This type of functions may be useful to print some messages, draw a set of lines, etc.

```
Eg: #include<stdio.h>
    displayline();
    message ();
    int main( )
        {
            displayline( );
            message( );
            displayline( );
            return 0;
        }

    displayline ( ) // no arguments and no return type for display line
   {
        int i;
        for(i=1; i<=80; i++)
            printf("-");
```

```
        printf("\n");
    }
    message( )
    {
        printf(" ** TRY TO UNDERSTAND THE LANGUAGE **** \n ");
        printf(" ***NOT TO BYHARD THE LANGUAGE ***** \n ");
    }
```

OUTPUT: -

```
        ** TRY TO UNDERSTAND THE LANGUAGE ****

        ***NOT TO BYHARD THE LANGUAGE *****
```

2. Functions *with* **arguments** and *without* **return values:**

- The nature of data communication between the calling function and the called function is with arguments but no return values.
- One-way data communication between calling function and called function through arguments.
- The control is transferred to the called function by passing some arguments, after performing the task; the control is back to the calling function without returning any value.

Eg:

```
#include<stdio.h>
displayline(char);
message (char);
int main( )
{
        char name[30] = "DENNIS RITCHIE";
        char ch = '-';
        displayline( ch);
        message( name);
        displayline( ch);
        return 0;
}
```

```
displayline (char x ) // with arguments and no return type for displayline
{
        int i;
        for(i=1; i<=80; i++)
                printf("%c", x);
        printf("\n");
}

message( char str[30])          // with arguments and no return type
{
        printf(" \t \t");
        printf(" %s", str);
        printf("\n");
}
```

OUTPUT: ----------

DENNIS RITCHIE

----

3. Functions *without* **arguments** and *with* **return values:**

- The function is called without passing any arguments, but after performing the task, it returns some value to the calling function.
- One-way data communication between calling function and called function through return statement.
- The control is transferred to the called function without passing any arguments, after performing the task; the control is back to the calling function by passing some value.

Eg:

```
#include<stdio.h>

 int factorial();

int main( )

{

        int f;
```

```
        f = factorial( );
        printf(" \n Factorial f = %d", f); return 0;
    }
int factorial( ) // without arguments, but returns some value.
    {
        int i, f =1,n;
        printf(" enter a value ");
        scanf("%d", &n); for(i=1; i<=n; i++)
            f= f * i ; return f;
    }
```

OUTPUT:- enter a value 4

Factorial f = 24

4. Functions *with* **arguments** and *with* **return values:-**

- Two-way data communication between calling function and called function is by passing arguments and sending back some value.
- The control is transferred to the called function by passing some arguments, after performing the task; the control is back to the calling function by returning some value.
- At the time of function calling the actual arguments are dumped into formal arguments.

Eg:

```
#include<stdio.h>
 int factorial (int x);
 int main( )
{       int n, f;
        int factorial(int);        // function declaration or prototyping printf("
        enter a value ");
        scanf("%d", &n);
        f = factorial (n);         // function calling statement
        printf(" \n Factorial f = %d", f);
```

```
        return 0;
}
int factorial( int x)    // function definition with argument, with return value.
{       int i, f =1;
        for(i=1; i<=x; i++)
                f= f * i;
        return f;
 }
```

Parameter passing techniques:

Parameter passing is a technique used to pass data to a function. Data are passed to a function using one of two techniques: **pass by value** (**call by value**) and **pass by reference** (call **by reference**).

1. Call by value:

- In call by value mechanism a copy of the data is sent to the function. That is **the values** of actual arguments are being **copied** into formal arguments. And ensures that the original data in the calling function cannot be changed.
- Memory is allocated temporarily for formal parameters and local variables.
- Whatever the modifications are done for formal parameters **will not affect** the actual parameters.

2. Call by Reference:

- In call by reference mechanism the address of the data rather than a copy is sent to the function. That is **the address** of actual arguments is being passed into formal arguments. The called function can change the original data in the calling function.
- When we pass the addresses, the receiving parameters should be pointers to hold these addresses.
- Whatever the modifications are done for formal parameters **will directly affect** the actual parameters.
- C language does not have a true call by reference and it is stimulated by call by address.

Eg:// program for **call by value** mechanism

```
#include<stdio.h>
int main( )
{
        int a, b;
        void swap(int, int);// function declaration or prototyping
         printf(" enter two values ");
        scanf("%d %d", &a, &b);
        swap(a, b);                    // function calling statement
        printf(" Values after swap function call are: ");
        printf(" a= %d \t b = %d ", a, b);
        return 0;
}
void swap( int x, int y)          // function definition.
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

OUTPUT:- enter two values 10 20

Values after swap function call are: a = 10 b=20

Here the **copy of the actual data** a, b is sent to formal parameters of swap function, in swap function these are swapped but the changes made to x, y are **not affect** the values of a, b in main function. Because the copy the data is sent but not original location.

Eg: // program for **call by reference** mechanism

```
#include<stdio.h>
int main( )
{
        int a, b;
        void swap(int, int); // function declaration or prototyping
        printf(" enter two values ");
        scanf("%d %d", &a, &b);
        swap(&a, &b); // function calling statement
        printf(" Values after swap function call are: ");
        printf(" a= %d \t b = %d ", a, b);
        return 0;
}
void swap( int *x, int *y)              // function definition.
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

OUTPUT:- enter two values 10 20

Values after swap function call are: a=20 b=10

Here the **address of the actual data** (&a, &b) is sent to formal parameters (*x, *y) of swap function, in swap function instead of creating temporary memory locations for x and y, the same memory locations of a, b referenced by x, y. So changes made to x, y are directly **affects a, b** in main function. So finally swapped values are stored in a, b .

Scope - Local and Global variables

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,
```

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 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;
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference

```
#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 with-in a function and they take precedence over global variables

Storage classes in C:

All the variables in C, not only to have a data type, they also have a "**Storage classes".** The storage class of variable tells the compiler that:

(a) **The storage area of the variable**: there are basically two kinds of locations in a computer where such variables or values to be kept, "Memory and CPU Registers. The storage class of a variable determines in which of these two locations the value is stored.

(b) **The initial value of the variable** if not initialized

(c) **The scope of the variable(active)** that in which functions the value of the variable would be available.

(d) **Life of the variable(alive)** that is how long the variable would be active or exist in the program, i.e the ***longevity*** of the variable.

There are *four* storage classes in C:

1. Automatic variables.
2. Register variables.
3. Static variables.
4. External variables.

1. **Automatic storage classes variables**: The features of automatic storage classes variables are :

   Storage **:** Memory

   Initial value: Garbage value or unpredictable value.

   Scope **: Local** to the block in which it is defined.

   Life time **:** Until end of function or end of block where it is defined.

1. Automatic variables are declared inside a function in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited.

2. Automatic variables are **local** to the function.
3. We use the keyword **auto** to declare automatic storage classes.
4. The **default** storage class of a variable is **automatic** storage class. Ex: int x ; it implies that auto int x;
5. Because of the scope of the automatic variables, they are also referred to **local** or **internal** variables.

Ex: 1.
```
main ( )
{
  auto int i; int j;
  printf ("\n %d \t %d", i, j);
}
```

Output: some garbage values, because they are not initialized to any value. A variable j without having any storage class is by default automatic. So both i, j are automatic variable.

Ex: 2.
```
main ( )
{
        auto int a = 1;
        {
            auto int a = 2;
        {
             auto int a = 3;
              printf("a = %d", a);
        }
               printf(" \n a = %d", a);
        }
        printf(" \n a = %d", a );
}
```

Output:
```
3
2
1
```

2. Register variables:

The features of a variable defined to be of Register storage class are:

Storage : Registers

Initial value : Garbage value or unpredictable value.

Scope : **Local** to the block in which it is defined.

Life time : Until end of function or block.

1. A variable stored in CPU Register can always be accessed faster than the one which is stored in memory.
2. If a variable is used at many placed in a program, then it is better to declare it as register variable. Ex:- loop counter variables.
3. We can use the keyword "**register"** to declare register variables.
4. We cannot use register class for all types of variables, because CPU registers in a micro computer are usually 16- bit registers and therefore cannot hold a float value or a double value which require 4 and 8 bytes respectively. However C will automatically convert register variables into non register variables once the limit is reached.

Ex:- main ( )

```
{
 register int i;
 for(i = 1; i < = 10; i++)
   printf("\n %d", i);
}
```

. **Static variables:** The features of static storage class variable are:

Storage : Memory

Initial value : Zero

Scope : Global (may be internal static or external static)

Life time : Value of the variable **persists** between different function call.

1. Static variables persist until the end of the program.
2. A static variable may be either an initial type or an external type, depending on the place of declaration.
3. We can use the keyword **"static"** to declare a variable as static variable.
4. A static variable is initialized only once.
5. The difference between auto variables and static variables don't disappear when the function is no longer active. These values persist, but auto variables disappear when the function is no longer active.

Ex:-
```
main ( )
{
  int i;
  for(i = 1; i < = 3; i++)
        Display ( );
}
Display ( )
{
  static int x = 0; x = x +1;
  printf("x = %d \n", x);
}
```

Output:-
```
x = 1
x = 2
x = 3
```

4. **External variables:-** The features of a variable whose storage class has been defined as external are as follows:

Storage	**:** Memory
Initial value	: Zero
Scope	**:** Global
Life time	**:** As long as the program's execution doesn't come to an end.

1. External variables are declared outside of the all functions.
2. The external variables are both alive and active throughout the entire program.
3. They may be declared using the keyword **"extern".**
4. The external variables are also known as global variables. These variables can be accessed by any function in the program.
5. In case, both external and auto variables are declared with the same name, in a program the first priority is given to the auto variable.
6. Use external storage class only these variables which are being used by almost all the functions in the program. This would avoid unnecessary passing of the variables as arguments when making a function call.

Ex:-
```
int i = 10;
 main ( )
{
 int i = 20;
 printf("\n %d",i);
 display ( );
 {
printf ("\n %d", i)
 }
```

Output: 20
10

Recursion:

- When a function *calls itself*, then that process is called as **recursion** and that function is called as recursive function.
- Recursive functions can be used to solve problems where the solution is expressed in terms of successively applying the same solution to subsets of the problems.
- Every recursive functions must has to have two basic properties:

(i) A termination condition called ***anchor step*** (usually if statement) to avoid infinite process.

(ii) A repetition statement called ***recurrence step*** to repeat the function calling process.

- Recursion is of two types. **Direct recursion** and **indirect recursion**.

- When a function calls itself, then it is *direct recursion*. But when function-1 calls function-2 and in turn function-2 calls function-1 then it is *indirect recursion*.

Eg: To calculate the factorial value of number 4, the recursion process is as follows.

```
int main( )

        {
                int n, f;
                int factorial(int);              // function declaration or prototyping
                n = 4;
                f = factorial (n);               // function calling statement
                printf(" \n Factorial f = %d", f);
                return 0;
                }

  int factorial( int x)    //function definition with argument, with return value.
 {
                int f;
                if( (x = = 1) || (x = = 0))
                      return 1;
                else
                return (x * factorial ( x - 1) );
  }
```

Anchor step: if((x = = 1) || (x = = 0))

Recurrence step: return (**x * factorial (x - 1))**

x * factorial (x - 1) is processed as follows:

4 * factorial(3)

3 * factorial (2)

2 * factorial (1)

1

4 * 3 * 2 * 1 = 24

Passing Arrays as a parameter to a function

An array can be passed as a parameter to a function.

Syntax: return type function_name(datatype *array_pointer);

or

return type function_name(datatype array_name[size]);

Example: void sum(int *a);

Where a is a formal parameter which holds the starting address of the array passed as a parameter to the function sum

Or

void sum(int a[10]);

Program for passing an array as argument to a function

```
#include<stdio.h>
void sum(int *a, int n);
int main( )
{
        int a[10], i, n;
      printf("Enter the size of the array");
        scanf("%d", &n);
      printf("Enter %d elements",n); for(i=0;
        i<n; i++)
            scanf("%d",&a[i]); sum(a,n);
}
void sum(int *a, int n)
{
    int i, sum = 0;
    for(i=0; i<n; i++)
        sum += a[i];
   printf("Sum : %d", sum);
}
```

Pointers to Functions

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

A pointer to a function can be declared as follows.

Syntax: datatype (*fptr)();

fptr is a pointer to a function which returns type value

A function pointer can be made to point to a specific function by assigning the name of the function to the pointer

Example:

int multiply(int a, int b);	function declaration
int (*fptr)(int, int);	pointer to a function
fptr = multiply;	assigning address of multiply to fptr
n=(*fptr)(a,b);	function call using pointer fptr

Program showing Pointer to a Function

```
#include<stdio.h>
int multiply(int a, int b);
{
    return a * b;
}
int main( )
{
    int a, b, c;
     int (*fptr)(int,int); printf("Enter two
    numbers"); scanf("%d%d",&a,&b);
    fptr = multiply; c =
    (*fptr)(a,b);
     printf("Result : %d",c);
}
```

Dynamic Memory Management

Dynamic allocation: Dynamic memory allocation is the process of allocating memory during execution time.

Dynamic Memory Allocation Functions in C:

This allocation technique uses predefined functions to allocate and release memory for data during execution time.

Function	Syntax
malloc ()	Void * malloc (size_t);
calloc ()	Void * calloc (int number, size_t);
realloc ()	Void * realloc (pointer_name, size_t);
free ()	free (pointer_name);

1. **malloc( ) :**
   - malloc( ) function is used to allocate space in memory during the execution of the program.
   - It does not initialize the memory allocated during execution.
   - It carries garbage value.
   - It returns null pointer if it couldn't be able to allocate requested amount of memory.
2. **calloc( ) :** calloc( ) function is also like malloc( ), but calloc( ) initializes the allocated memory to zero.
3. **realloc( ) :**
   - 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.
4. **free ( ) :**
   free ( ) function frees the allocated memory by malloc ( ) and calloc ( ) functions and returns the memory to the system

Example:

Program to allocate memory for an array dynamically, store and access its values

```
#include<stdio.h>
#include<stdlib.h>
 int main( )
{
     int *a, i, n;
     printf("Enter the size of the array");
     scanf("%d",&n);
     a = (int *) calloc(n , sizeof(int));
     printf("Enter %d elements into the array");
     for(i=0; i<n; i++)
          scanf("%d",&a[i]);
          printf("\nElements of the array");
     for(i=0;i<n;i++)
        printf("%d ",a[i]);
         free(a);
}
```

Difference between static memory allocation and dynamic memory allocation

Static Memory Allocation	Dynamic Memory Allocation
It is the process of allocating memory at compile time.	It is the process of allocating memory during execution of program
Fixed number of bytes will be allocated	Memory is allocated as and when it is needed.
The memory is allocated in memory stack	The memory is allocated from free memory pool (heap).
Execution is faster	Execution slow
Memory is allocated either in stack area or data area	Memory is allocated only in heap area
Ex: Arrays	Ex: Dynamic arrays, Linked List, Trees

UNIT – IV

Topics going to covered:

Structures and Unions - Declaration, initialization, accessing structures, operations on structures, structures containing arrays, structures containing pointers, nested structures, self referential structures, arrays of structures, structures and functions, pointers to Structures**,** unions.

Structure is a mechanism for packing data of different types. It is a convenient tool for handling a group of logically related data items. Simply the structure is a collection of logically related elements of different data types.

Eg: It can be used to represent a set of attributes, such as student-name, roll-number, and marks. We use a structure to represent all these different data elements.

Declaration of a structure: A structure is declared as follows:

Syntax:

```
struct   tag-name
{
Data type      member-1;
Dat type       member-2;
:
:
Data type      member-n;
};
```

Eg:

```
struct   student
```

```
{
        int     rno;
        char    name[20];
        int     tm;
        flaot   avg;
};
```

- Here "struct" is a keyword used to declare a structure.
- Tag-name is the name given to structure declared.
- Member-1, member-2,….member-n are elements or fields of the structure, each member may belong to different data type.
- Structure declaration should be terminated by semicolon.
- Structure declaration doesn't get any space for its members, it simply describes a format called template to represent the information.
- Memory allocated only after declaring structure variables.
- Structure variables can be declared using tag-name as follows:

Syntax: struct tag-name variable-1, variable-2, variable-N;

Eg: struct student s1, s2, s3;

Here student is a structure and it is having 4 different members which are logically related. And s1, s2 and s3 are three structure variables.

- It is also allowed to combine both the template declaration and structure variables declaration in one statement. The declaration is as follows:

```
struct student
{
        int     rno;
        char    name[20];
        int     tm;
```

```
        flaot   avg;
}s1,s2;
```

- Tag-name is optional in structure declaration. When tag-name is absent then structure variables should be created in structure declaration itself, before semicolon.

Eg:

```
struct
{
        int     rno;
        char    name[20];
        int     tm;
        flaot   avg;
} s1, s2;
```

- Individual members of the structure cannot be initialized within structure declaration.

```
struct  student
{
        int     rno = 28;
        char    name[20] = "SUN";          Invalid
        int     tm = 500;
        flaot   avg = 90;
};
```

Accessing the members of the structure:

Members of the structure can be accessed by writing structure variable name with member selection operator (dot operator **.**) and member name.

structure variable **.** member; Eg:

s1. rno = 28; s1 .

tm = 500;

Structure Initialization:

Like any other data type, a structure variable can be initialized. We can initialize a structure at compile time and also we can initialize the structure members at run time.

- **Compile time initialization:** One way to initialize structure members is direct initialization.

  struct student s1 = { 28, " SUN", 500, 90 };

  Here, when initializing the structure directly like above, and then the order should be maintained. That is the order of assigning values is same as the order of members declared in the structure.

- And other way to initialize the structure members as follows:

  struct student s1;

  s1.rno = 28;

  strcpy (s1.name,"SUN");

  s1.tm= 500;

  s1.avg = 90;

- we can initialize structure members at run time by considering the scanf statement as follows: scanf ("%d%s%d%f", &s1.rno, s1.name, &s1.tm, &s1,avg);

```
/* Example for structure declaration and initialization */

#include<stdio.h>
 struct student
{
int rno;

char name[20];
 int tm;
float avg;

};
```

```
void main()
{
struct student s1 = { 28,"SUN",450,89};
 clrscr();
printf("\n Roll Number : %d",s1.rno);
printf("\n Name : %s",s1.name);
printf("\n Total : %d",s1.tm);
printf("\n Average: %f", s1.avg);
 getch();
}
```

Output:-

Roll Number: 28 Name:
SUN Total: 450
Average: 89.000000

Comparison and Copying of structure variables:

- We can copy one structure variable into another; it can be achieved by copying individual members or entire structure at a time by using assignment statement.

Eg: struct student s1, s2; s2 = s1;

- Comparison of structure variables for their equality and inequality purpose by using equality operator (==), and inequality operator (!=) is not allowed in all compilers. If it is necessary to compare two structure variables then it is to follow individual structure member's comparison

Arrays of structures

- We can declare an array of structures, each element of the array representing a structure variable. Eg: The total marks and average marks analysis of the marks report obtained by a class of 60 students is as follow:

```
struct  student  s[60];
```

Here s is an array of 60 elements to the structure student, or simply s is an array of structures.

- Array of structures allowed us to create an array of similar type of elements which themselves are a collection of dissimilar data type.

In an array of structures all elements of the array are stored in adjacent memory locations

Eg /* Calculating the total marks of 3 students */

```
struct   marks

{

int      m1;

int      m2;

int      m3;

int      m4;

};

int main( )

{

int sum=0, i;
struct   marks s[3] = { {56,67,78,59}, {60,50,70,80}, {52,62,72,82} };

for(i=0;i<3;i++)

{

sum=0;

sum = sum + s[i].m1 + s[i].m2 + s[i].m3 + s[i].m4;
printf("\n Total marks of %d = %d", s[i], sum);
}

return 0;
                        }
```

Arrays within structure

C language allows us to use arrays as structure members. We can use arrays as structure members as single dimensional or multi dimensional arrays, or array of characters.

```
Eg:-    struct   student

        {

                char     name[20];

                int      rno;
```

```
        int     m[4];

        int     tm;

    };
```

Here the structure members name and m are arrays, where name is character array and m is integer array. The variable name contains 20 elements. And m contains 3 elements m[0],m[1],and m[2], these represent marks obtained by a student in 3 subjects. These elements can be accessed using appropriate subscripts.

/* Program to read student name, roll number and marks in 5 subjects and calculate total marks and average marks obtained by 3 students using array of structures and arrays with in structures */

```
struct  student
{
        char sname[20];
        int rno;
        int m[5];
        int tm; float
        avg;
}s[3];

int main()
{
     int i,j,t;
    for(i=0;i<3;i++)
   {
        printf("\n Enter %d student details \n ",i+1);
        printf("\n enter student name and roll number ");
        scanf("%s%d",s[i].sname, &s[i].rno);
        printf("\n Enter marks in 5 subjects ");
        for(j=0;j<5;j++)
        {
                scanf("%d",&s[i].m[j]);
```

```
            }
    }
    for(i=0;i<3;i++)
    {
            t=0;
            for(j=0;j<3;j++)
            {
                    t=t+s[i].m[j];
            }
            s[i].tm=t;
            s[i].avg=s[i].tm/5;
    }
    printf("\n Students details are \n");
    for(i=0;i<3;i++)
    {
            printf("\n Name: %s", s[i].sname);
            printf("\n Number : %d",s[i].rno);
            printf("\n Total : %d",s[i].tm);
            printf("\n Average : %f", s[i].avg);
    }
    return 0;
}
```

Structures and Pointers:

- ✓ Pointer as a structure variable
- ✓ Pointers within structures

Pointer is a variable that holds address of another data variable. We can also define pointer to structure. That is we can declare a pointer as a structure variable. Here starting address of the structure member variables can be accessed to pointer variable of he same structure.

Eg:
```
struct book
{
   char name[20];
   char author[20];
    int pages;
};
```

struct book *ptr;

Here ptr is a pointer variable to the structure book. We can access the members of the structure by using ptr variable and is accomplished by indirect member selector operator (->) or structure pointer operator.

i,e ptr->name, ptr->author, ptr->pages.

```
/*Example for pointer to structure*/

struct book
{
   char name[20];
   char author[20];
    int pages;
};
int main()
{
struct book b1 = {“let us c”,”kanetkar”,100};
 struct book *ptr;
ptr=&b1;
printf(“\n%s by %s of %d pages”,b1.name,b1.author,b1.pages);
printf(“”\n %s by %s of %d pages”, ptr->name, ptr->author, ptr->pages);
return 0;
}
```

In the above example ptr is a variable, which is used to hold the address of structure variable. The

structure members can be accessed using the indirect selector operator.

*ptr -> rno is same as (*ptr).rno.

The parenthesis around *ptr are necessary, because the member operator, has a higher precedence than the operator “*”.

Pointer as structure member:

We can take pointer as a structure member. We can access these members through structure variable

Eg: /*Pointer as structure member*/

```
int main()
{
int a=21, h=5.40;
 struct boy
{
char *name;
int *age;
float *height;
};

struct boy b,*ptr;

strcpy(b.name,"SUN");
 b.age=&a;
b.height=&h;
printf("\n Name :%s",b.name);

printf("\n Age :%d",*b.age);

printf("\nHeight :%d",*b.height);
ptr=&b;
printf("\n through pointer, accessing the structure variable");
 printf("\n Name          :%s",ptr->name);
printf("\n age    :%d",*ptr->age);
printf("\nHeight          :%f",*ptr->height);
 return 0;
}
```

Here **b** and ***ptr** are also two structure variables. The members of structure variable boy are pointers. The pointer *ptr is pointer to structure boy, We can access members of structure boy through b and ptr, and by using direct selector (**.**), indirect selector (**->**). The members can be accessed through the variable b is as follows:

b.name *b.age *b.height;

The members can be accessed through the variable ptr is as follows.

ptr->name, *ptr->age, *ptr->height.

If **p** is a structure variable and **age** is a member of the structure then the following are different notations to access the member in different ways.

p . age **:** when both p and age are ordinary variables (use direct member selector **.**)

p -> age : when p is a pointer to the structure and age is ordinary member of the structure(use indirect member selector ->).

***p . age** : when p is ordinary variable to structure and age is a pointer member of the structure.

***p -> age :** when both p and age are pointer variables

Structures and Functions:

Like variables of standard data type, structure variables can also be passed to the function by value or address. Whenever a structure element is passed to declare the structure variable outside the main() function i.e. global.

- We can pass individual elements of a structure variable as parameters to functions
- We can pass an entire structure as an argument.
- We can also pass address of the structure as an argument to a function.

i) Passing members of a structure as arguments to a function.

```
struct boy
{
  char name[20];
  int age;
  int wt;
};
struct boy b1={“amit”,20,45};
main()
{
   display(b1.name,b1.age,b1.wt);
}
display(char s[20],int a,int w)
{
printf(“\n Name :%s”,s);
printf(“\n Age :%d”,a);
printf(“\n weight :%d”,w);
}
```

In the above example, individual elements of structure boy (members of structure) are passed as arguments to the function display(). b1.name, b1.age, b1.wt are actual arguments s, a and w are formal arguments. When a function call is made, actual arguments values are copied into formal arguments.

ii) Structure as argument to the function.

```
struct boy
{
  char name[20];
  int age;
   int wt;
};
main()
{
    struct boy={“amit”,20,45};
    display();
};
void display(struct boy y)
{
     printf(“\n Name : %s”, y.name);
     printf(“\n Age :%d”,y.age);
     printf(“\n weight %d”,y.wt);
}
```

In the above example boy is a structure, b1 is structure variable. By passing the entire structure as an argument to the function display( ),printing the values. Actual argument is b1 and formal argument is y.b1 values are dumped into y, when a function call is made.

i) passing the address of a structure as an argument to the function.

```
struct boy
{
   char name[20];
    int age;
     int wt;
};
```

```
main()
{
   struct boy b1={“amit”,20,45};
    display(&b1);
}

void display(struct bot *y)
{
     printf(“\n name :%s”,y-name);
     printf(“\n age :%d”,y_->.age);
     printf(“\n weight :%d”,y->wt);
}
```

In the above example the address of structure boy(&b1) is passing as an argument to the function display(),printing the values. Actual argument is&b1, formal argument is *y; When we are passing the address, the receiving parameter should be pointer. We can access the members of a structure through pointer variable to that structure by using indirect selector(->) operator.

We can also consider that structure as return type of a function. The following example illustrates structures with functions as arguments and return type.

/*program for addition, subtraction, multiplication and division of two complex numbers using structures and functions*/

```
struct complex
{
  float real; float
  imag;
};
struct complex a, b, c;
int main()
{
int ch;
printf(“enter first complex no:”);
scanf(“%f%f”,&a.real, &a.imag);
printf(“enter second complex no:);
```

```
            scanf("%f%f",&b.real, &b.imag);
            display(a);
            display(b);
            printf(" MENU \n");
            printf("\n1.addition");
            printf("\n2.subtraction");
            printf("\n3.multiplication");
            printf("\n4.division");
            printf("\n Enter ur choice:");
            scanf("%d", &ch); switch(ch)
            {
            case 1:c=add(a,b);
            break;
            case 2:c=sub(a,b);
            break;
            case 1:c=mul(a,b);
            break;
            case 1:c=div(a,b);
            break;
             default: printf("\n your choice is invalid");
}
display(c);
}

struct complex add(struct complex x, struct complex y)
{
struct complex t;
 t.real=x.real+y.real;
 t.imag=x.imag+y.imag;
return t;
}
struct complex sub(struct complex x, struct complex y)
{
struct complex t;
t.real=x.real-y.real;
 t.imag=x.imag-y.imag;
return t;
}

struct complex mul(struct complex x, struct complex y)
{
struct complex t;
t.real=x.real*y.real-(x.imag * y.imag);
t.imag=x.imag*y.imag+(x.imag*y.imag);
```

```
return t;
}
struct complex div(struct complex x, struct complex y)
{
struct complex t; float d;
d=(y.real*y.real)+(y.imag*y.imag);
t.real=((y.real*y.real)+(y.imag*y.imag))/d;
 t.imag=((y.real*y.imag)-(y.imag*y.real))/d;
return t;
}
display(struct complex c)
{
printf("\n\t\t %f + i %f", c.real, c.imag);
}
```

Structures within structures

Structures within structures mean nesting of structures. Nesting of structures permitted n C. We can take any data type variable a structure member, like that we can also take object of one structure as member in another structure. And we can group all the items related to one particular object is also declared as structure member.

```
/*Example for structures within sructures*/

struct name

{

char *first; char *last;
};

struct emp

{

struct name n1; int eno;
float sal;

};

main()

{

struct emp e;
```

```
printf("enter employee details");
 printf("\n Name : first,last");
printf("\n employee no and sal");

scanf("%s%s%d%f",e.n1.first,e.n1.last,&e.eno,&e.sal);
printf("\n Name : %s",strcat(e.n1.first,e.n1.last);
printf("\n empno :%d",e.no);
printf("\n salary:%f",e.sal);
}
```

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

In other words, structures pointing to the same type of structures are self-referential in nature.

Example:

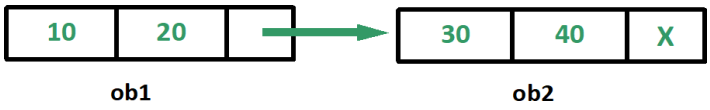
```
struct node {
        int data1; char
        data2;
        struct node* link;
};

int main()
{
        struct node ob; return
        0;
}
```

In the above example 'link' is a pointer to a structure of type 'node'. Hence, the structure 'node' is a self-referential structure with 'link' as the referencing pointer. An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self-Referential Structures

- Self Referential Structure with Single Link
- Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.

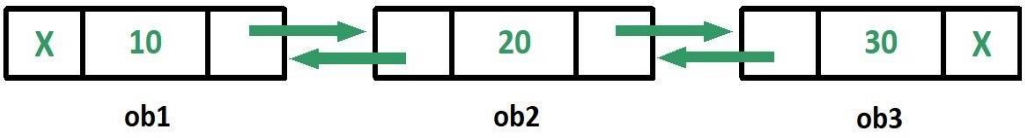


Example:

```
#include <stdio.h>

struct node {
        int data1; char
        data2;
        struct node* link;
};

int main()
{
        struct node ob1; // Node1

        // Initialization
ob1.link = NULL;
 ob1.data1 = 10;
ob1.data2 = 20;

struct node ob2; // Node2

// Initialization
ob2.link = NULL;
ob2.data1 = 30;
ob2.data2 = 40;

// Linking ob1 and ob2
ob1.link = &ob2;


// Accessing data members of ob2 using ob1
printf("%d",  ob1.link->data1);
printf("\n%d", ob1.link->data2);
return 0;

output:
30
40
```

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using

these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.The connections made in the above example can be understood using the following figure.



Example:

```
#include <stdio.h>

struct node {
        int data;
        struct node* prev_link;
        struct node* next_link;
};

int main()
{
struct node ob1; // Node1

// Initialization
ob1.prev_link = NULL;
 ob1.next_link = NULL;
 ob1.data = 10;

struct node ob2; // Node2

// Initialization
ob2.prev_link = NULL;
 ob2.next_link = NULL;
 ob2.data = 20;

struct node ob3; // Node3

// Initialization
ob3.prev_link = NULL;
ob3.next_link = NULL;
ob3.data = 30;

// Forward links
 ob1.next_link = &ob2;
 ob2.next_link = &ob3;
```

```
// Backward links
ob2.prev_link = &ob1;
 ob3.prev_link = &ob2;

// Accessing data of ob1, ob2 and ob3 by ob1
 printf("%d\t", ob1.data);
printf("%d\t", ob1.next_link->data);
 printf("%d\n", ob1.next_link->next_link->data);

// Accessing data of ob1, ob2 and ob3 by ob2
 printf("%d\t", ob2.prev_link->data);
 printf("%d\t", ob2.data);
printf("%d\n", ob2.next_link->data);

// Accessing data of ob1, ob2 and ob3 by ob3
printf("%d\t", ob3.prev_link->prev_link->data);
 printf("%d\t", ob3.prev_link->data);
 printf("%d", ob3.data);
return 0;
}
```

Output:

```
10    20    30

10    20    30
10    20    30
```

In the above example we can see that 'ob1', 'ob2' and 'ob3' are three objects of the self referential structure 'node'. And they are connected using their links in such a way that any of them can easily access each other's data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

Applications:

Self referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs etc

Unions:

Union is a concept borrowed from structures and therefore follows the same syntax as structures. Union is a group of logically related data values of different type with the common name. And we can use . (dot) operator to access individual members of a union as in structures.

```
Ex:     union
        {       int m;
                float x; char
                c;
                }       code;   code.m=10, code.x=12.05, code.c='A';
```

However, there is major distinction between the union and structure is in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location. Union can handle only one member at a time.

Ex: structure, union declarations are as follows:

```
struct item1                    union item2
{                               {
    int m;                          int m;
    float x;                        float x;
    char c;                         char c;
}code1;                         } code2;
```

Differences between structure and union:

Structure	Union
Collection of values of different data types	Collection of values of different data types

Declaration:
struct <tag name>
{
Datatype1 member1;
Datatype2 member2;
.
.
.
.
Datatype-n members-n;
};
	Declaration:
	union <tag name>
	{
	Datatype1 member1;
	Datatype2 member2;
	.
	.
	.
	.
	Datatype-n members-n;
	};

The memory allocated for structure is equal to its members total size. (Memory is allocated only when a structure variable is defined).	The memory allocated for union is equal to its member size, which allocation (Memory is allocated only when a union variable is defined).
struct item1 { Int m; float x; char c; }s;	union item2 { Int m; float x; char c; }u;
Memory allocated for s is 2+4+1=7 bytes.	Memory allocated for u is 4 bytes
All the members can be accessed at a time.	Only member can be accessed at a time.
Values assigned to one member will not cause the change in other members.	Values assigned to one member May cause the change in value of other members.

The usage of structure is efficient when all Members are actively used in the program	The usage of union is efficient when members of it are not required to be accessed at the same time.

FILES

Topics covered:

Files – Concept of a file, Streams, Text files and Binary files, Basic operation on files, File input / output functions. Sequential Access and Random Access Functions

A **File** is a collection of records that can be accessed through the set of library functions. Record is nothing but collection of fields of related data items. These files are stored on the disk and can be accessed through file-handling functions provided by the C-standard library.

RollNo	Name	Total Marks	Average
101	Pavan	450	70
102	Laxman	400	66
103	Dinesh	433	68
104	Simha	500	80
105	Sun	540	82

Eg: Here a file called "STUDENT.DAT" which is having 5 students' records. And every record is collection of 4 fields.

Types of I/O: There are numerous library functions available for I/O. these can be classified into two broad categories:

a) Console I/O b) file I/O

Console I/O:Functions to receive input from keyboard and write output to VDU (monitor). The screen and keyboard are called a console. Console I/O functions can be further classified into two

categories – formatted console I/O and unformatted console I/O. The basic difference between them is that the formatted functions allow the input and output to be formatted as per requirements. The different console I/O functions are as follows:

Formatted functions: The input function scanf( ) and the output function printf( ) are called formatted console I/O functions. By using the format specifiers and escape sequence characters we can use these functions to read/print required format of information.

Unformatted functions:- The input functions like getch( ), getche( ), getchar( ) and gets( ) are called unformatted input functions and putch( ),putchar( ), and puts( ) are unformatted output functions. Each and every function is having its own syntax and meaning.

File I/O: Sometimes it is necessary to store the data in a manner that can be later retrieved and displayed either in a part or in whole. This medium is usually a "file" on the disk. File I/O can be handled by using different functions.

Formatted functions: The file input function fscanf( ) and the file output function fprintf( ) are called formatted file I/O functions.

Unformatted functions: The input functions like getc( ), getw( ), and fread( ) are called unformatted file input functions and putc( ), putw( ), and fwrite( ) functions are unformatted file output functions. Each and every function is having its own syntax and meaning.

File streams:- Stream is either reading or writing of data. The streams are designed to allow the user to access the files efficiently. A stream is a file or physical device like key board, printer, monitor, etc., The FILE object uses these devices. When a C program is started, the operating system is responsible for opening three streams: standard input stream (**stdin**), standard output

stream (**stdout**), standard error(**stderr**).Normally the stdin is connected to the keyboard, the stdout and stderr are connected to the monitor.

Text files and Binary files: C uses two types of files, text files and binary files.

Text files:- Text file is a file consists of a sequence of characters divided into lines with each line terminated by a new line('\n'). A text file is writing using text stream. We can read and write text files

using different input/output functions. Formatted input/output (scanf/printf), character input/output (getchar/putchar), and string input/output (gets/puts) functions. And these functions can work with only text files. A text file contains only textual information like alphabets, digits, and special symbols. In actuality the ASCII codes of these characters are stored in text files.

Ex: A good example of text file is any C program file.

Binary files: A binary file is a collection of data stored in the internal format of the computer. Unlike text files, the data do not need to be reformatted as they are read and write rather, the data is stored in the file in the same format that they are stored in memory. Binary files are read and write using binary streams known as block input/output functions. Simply a binary file is merely a collection of bytes. This collection might be a compiled version of a C program or music data stored in a wave file or a picture stored in a graphic file.

Differences between Text and Binary files:

The major characteristics of text files are:

1. All data in a text file are human-readable graphic characters.
2. Each line of data ends with a newline character.
3. There is a special character called end-of-file (EOF) marker at the end of thefile. The

major characteristics of binary files are:

1. Data is stored in the same format as that is stored in memory.
2. There are no lines or new line characters.
3. There is an end-of-file marker.

Basic operations on files: The file consists of large, amount of data, which can be read or modified depending on the requirement. The basic operations that can be performed on files are:

a) Opening a file: A file has to be opened before to read and write operations. This can be achieved through fopen() function. **Syntax:** FILE *fp;

```
fp=fopen(filename,mode);
```

Here is a pointer pointing to the file "filename", which can be opened in specified mode. The

fopen() performs three tasks:

i) It searches the disk for opening the file .

ii) If file exists, it opens that file, if the file is not existing this function returns NULL in case of read mode. In case of write mode, it creates a new file and in case of append mode it opens that file for updating.

iii) It locates a file pointer pointing to the first character of the file.

Ex: FILE *fp;

```
fp = fopen("sample.txt", "r"); if
(fp==NULL)
printf("file does not exist"); else
{
  .....
}
```

b) Reading/writing a file: Once the file is opened, the associated file pointer points to the starting of the file. If the file is opened in ***writing mode***, then we can write information by using different functions like:

putc ( ): Putting a character in to the file. It works with only character data type. One character at a time can write into a file.

Ex: char ch ='a'; putc (ch, fp);

putw ( ): putting or writing of an integer value to a file.

Ex:

```
int x = 5;
putw(x,fp);
```

fprintf(): writing a group of values into a file.

Syntax: fprintf (file pointer, ”control string”, list of arguments); Ex:

fprintf (fp, ”%s %d %d ”, name, rno, marks);

If the file is opened in ***reading mode***, then we can read information from the file, i.e. using some functions like:

getc ( ): getting a character from the file, or reading the file information character by character at a time, upto the end of the file by using this function.

Ex: char ch;

ch = getc (fp);

getw ( ): getting or reading integer value from a file. Ex:

int x;

x = getw (fp);

fscanf ( ): This function is used to read the information from a file record-wise. It is used to read more values at a time.

Ex: fscanf(fp,”%s%d%d”, name, &no, &marks);

Among all the above different file i/o functions, fscanf() and fprintf() are called as formatted i/o functions. Printf and scanf are also called as formatted i/o functions.

NOTE: some compilers refer that: putc()

is same as fputc()

getc() is same as fgetc().

c) closing a file: After reading/writing a file ,it is needed to close that file .fclose() is used to close a file. It closes only one file at a time.

fclose (fp):-close a file which is pointed by fp.

fcloseall ( ):-close all opened files at a time.

Different modes to open a file: The tasks performed by fopen() function when a file is opened in each of these modes are as follows:

1.**"r" (Read) mode**: opens the file that already exists for reading only. If the file doesn't exist it returns NULL.

2.**"w"(write) mode**: File is opened for writing, if the file exists its contents are overwritten and if the file doesn't exists, then a new file is created. It returns NULL if it is unable to open the file.

```
f1=fopen("sample.txt", "w");
```

3.**"a"(append) mode**: searches for the file, if it exists then appending new contents at the end of the file.If a file doesn't exist then a file with a specified name is created and ready to get append or add.

```
f1 = fopen ("sample.txt", "a");
```

4."**r+" (Read & write mode):** This is for both reading and writing the data. If the file doesn't exist then it returns NULL.

5."**a+" (Append & Read) mode**: The file can be read as well as data can be appended.

The two modes of binary files are:

1. **"rb" (read binary) mode:** Binary file is opened for reading only.
2. **"wb" (write binary) mode:** Binary file opened for writing only.

NOTE: While opening the file in text mode we can use either "r" or "rt"/ "w" or "wt", but since text mode is the default mode we usually drop the "t".

```
/* PROGRAM TO COPY ONE FILE INTO ANOTHER FILE */

#include <stdio.h>
int main()
{

FILE *fs,*ft; char ch;
fs=fopen("sample.txt","r");
if (fs==NULL)
{

puts("cannot open source file\n");
exit(1);
}

ft=fopen("sample1.txt","w");
 if (ft==NULL)
{

puts("cannot open target file");
 exit(1);
}

while((ch=fgetc(fs))!=EOF)
 fputc(ch,ft);

fclose(fs);
 fclose(ft);
return 0;
}
/* EXAMPLE PROGRAM FOR ACCESSING BINARY FILES USING fwrite( ) , fread( )

FUNCTIONS */

#include<stdio.h>
int main()
{
int i,n;
FILE *fp1,*fp2,*fp3;
 struct employ
{
char name[20];
 int eno;
int bsal;
};
struct employ e[3];
fp1=fopen("EMP.DAT","wb");
for(i=0;i<3;i++)
{
printf("\n Enter name, num, and basic pay \n");
```

```
scanf("%s%d%d",e[i].name,&e[i].eno,e[i].bsal);
}
fwrite(&e,sizeof(e),3,fp1);
fclose(fp1);

fp1=fopen("EMP.DAT","rb");
fread(&e,sizeof(e),3,fp1);
 for(i=0;i<3;i++)
{
printf("\n Name:%s",e[i].name);
printf("\n Number        :%s",e[i].eno);
printf("\n basic pay :%s",e[i].bsal);
}
fclose(fp1);
return 0;
}
```

Random Access to Files:

Files can be accessed either sequentially or randomly. Random accessing of files can be achieved by using some predefined C library functions: ***fseek( ), ftell( ),*** and ***rewind( ).***

ftell( ) function:- If we wish to know where the file pointer is positioned right now, we can use the function ftell. ftell takes a file pointer as an argument, and it returns the position as **long int** which is an offset from the beginning of the file. It takes the following form:

syntax: n = ftell(fp);

n would give the relative offset(in bytes).

rewind( ):- This function places the file pointer to the beginning of the file, irrespective of where it is present right now. It takes file pointer as an argument.

Syntax: rewind(fp);

fseek( ) function:- The fseek( ) function lets us move the pointer from one record to another. Simply it is to move the file pointer to the desired location within a file. When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and fseek returns -1. It takes the following form:

syntax: fseek(file pointer, offset, position);

- *file pointer* is a pointer to the concerned file.

- *Offset* is a number or variable of type long, it specifies the number of positions (bytes) to be moved from the location specified. If offset is positive number, then moving forward or negative meaning move backwards.
- *Position* is a n integer number and it specifies from which position the file pointer to be moved. Position can take one of the following three values.

  0 beginning of file

  1 current position

  2 end of file

Eg: fseek (fp, 0L,0);- go to the beginning of the file. (Similar to rewind).

fseek (fp, 0L,1);- Stay at current position (Rarely used)

fseek (fp, 0L,2);-go to the end of the file, past the last character of the file. fseek (fp,

m,0);-move to the (m+1)th byte in the file

fseek (fp, m,1); - move to forward by m bytes from current position fseek (fp,

-m,1); - move to backward by m bytes from current position fseek (fp, m,2); -

move backward by m bytes from the end.

/* PROGRAM TO ACCESS A FILE RANDOMLY */

```
#include<stdio.h> int
main()
{
    FILE *fp;
    long n; char c;
    fp = fopen("sample.txt", "w+");        // opened for both writing and reading
        while((c=getchar())!=EOF)    // press CTRL+Z after entering the text
        {
```

```
            putc(c,fp);
        }
        n=ftell(fp);                    // stores the present location of file pointer in n
        printf("\n No.of characters entered = %ld \n", n);
        n=0L;
        while(!feof(fp))
        {
            fseek(fp,n,0);              // moves file pointer to the beginning of the file.
            c=getc(fp);
            putchar(c);
            n=n+5L;
        }
        putchar('\n');
        return 0;
    }
```

The following program to illustrate the concept of command line arguments to file

```
/*PROGRAM THAT WILL RECEIVE A FILENAME AND A LINE OF TEXT AS COMMAND LINE ARGUMENTS AND WRITE
THE TEXT TO THE FILE */
int main(int argc, char *argv[] )

{

FILE *fp; int I;
char str[20];

fp = fopen(argv[1], ”w”);

printf(“\n The no.of arguments in command line are : %d \n ”, argc); for(i=2;i<argc;i++)
{
fprintf(fp,”%s”, argv[i]);

}

fclose(fp);

printf(“\n Contents of the file are :\n”);
 fp=fopen(argv[1],”r”);
 for(i=2;i<argc;i++)
```

```
{
fscanf(fp,”%s”, str);

printf(“%s”, word);

}

fclose(fp);

  return 0;
}
```

UNIT-V

BASIC DATA STRUCTURES

Topics covered:

Introduction to Data Structures – Linear and Non-Linear Structures – Implementation of Stacks, Queues, Linked Lists and their applications.

Introduction to Data Structures:

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called data structure. **(OR)**

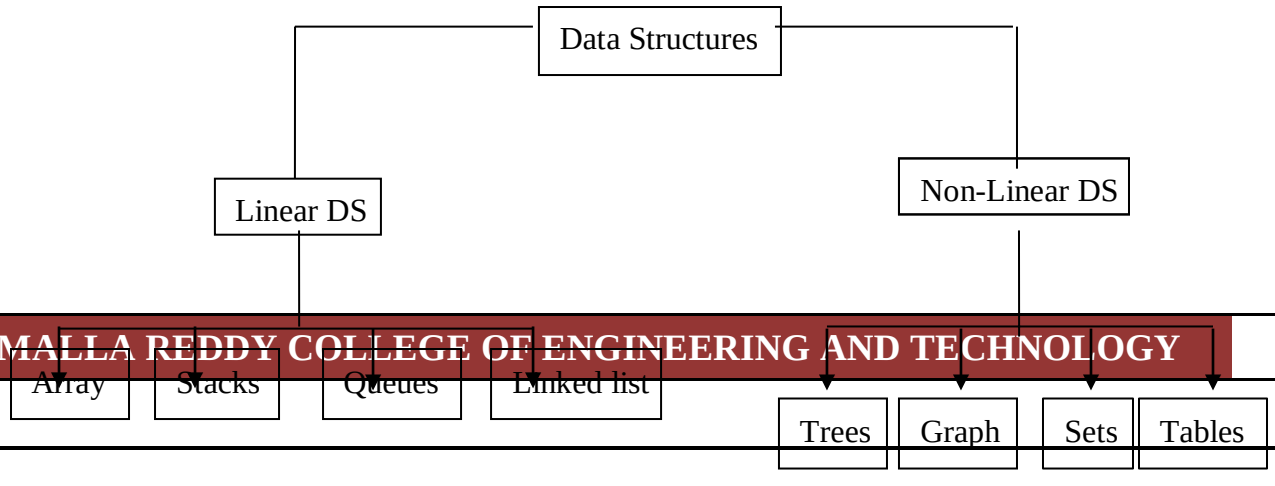
It is a study of different methods of organizing the data in memory and possible operations on these structures, and devices algorithms for these operations.

Data Structures = {Data items + storage methods + algorithms} The following are the activities associated with Data Structure

1. Data organization or clubbing
2. Accessing technique
3. Level of Associability
4. Manipulating selection for information
5. The Data structure should be satisfactory to represent the relationship between data elements.

Abstract Data Types

Data Structures is a group of items in which each item is defined by its own identifier, each of which is known as member of the structure. An useful data type for specifying the logical properties of a data type is the Abstract Data Type, or ADT. Fundamentally, a data type is a collection of values and a set of operations on those values. That collection and those form a mathematical construct that may be implemented using a particular hardware or software data structure. Data structure is also called as ABSTRACT DATA TYPES.



Linear Data Structure: Here all the elements form a sequence or maintain a linear ordering. **Non-Linear Data Structure:** These are the data structures where in the access and storage of data is not linear. Here all the elements are distributed over a plane.

Stacks:

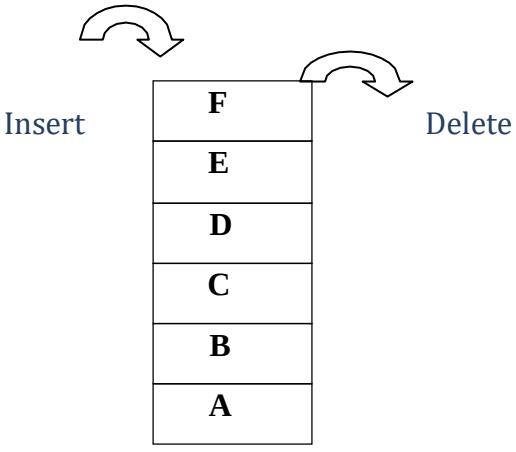
A stack is an ordered collection of data in which data is inserted and deleted at one end (same end) called as TOP.

- Stack is a linear data structures.
- The elements in a stack can be inserted and deleted at one end called top of the stack.
- Stack operates on a LIFO (last in first out) basis.
- Stacks are associated with two operations, they are push and pop.
- Stack is also known as pushdown list.

Push: To insert an element into the stack when we try to insert elements into the stack, if there is no space in the stack, then it is said to be OVERFLOW condition (stack is full) **Pop:** To delete an element from the stack. When we try to delete items from the stack if there is no item in the stack, then it is said to be in UNDERFLOW condition (stack is empty).

Example: Representation of Stack in memory

1. Pile of plates
2. A coin stacker



Stacks can be implemented by using

- Arrays (Static Implementation)
- Linked lists (Dynamic Implementation)

Algorithm for Stack operations using Arrays: Procedure

PUSH (A, MAX, TOP, ITEM)

A : Array

MAX : Stack size

TOP : Stack pointer

ITEM : value in a cell

Step 1: check for stack overflow If

TOP>= MAX-1 then

Printf" stack is OVERFLOW"

Return;

Step 2: Increment top

TOP= TOP + 1

Step 3: Insert item at the top of the stack A

[TOP] = ITEM

Return Step

4: Exit

Procedure for pop (A, TOP)

A : Array

TOP : Stack pointer

Step 1: check for stack UNDERFLOW If

TOP < 0 then

Printf" Stack is UNDERFLOW" and exit

Return

Step 2: Return former top element of stack ITEM

= A [TOP]

Step 3: Decrement pointer TOP TOP =

TOP – 1

Return Step

4: Exit

Stacks using Arrays

```
#include<stdio.h>
 #include<conio.h>
#define MAX 5
int top=-1;

int s[MAX]; void push(int x)
{
if(top==MAX-1)
printf("\n stack overflow"); else
{
top++; s[top]=x;
}
}
int pop( )
{
int y; if(top==-1)
{
printf("\n stack underflow"); return -1;
}
else
{
y=s[top];
top--; return y;
}
}
void display()
{
int i; if(top==-1)
printf("\n satck underflow"); else
```

```
{
printf("\n stack elements are: ");

for(i=top;i>=0;i--)
printf("%4d",s[i]);
}
}
int main()
{
int ch,a,b;
 while(1)
{
printf("\n stack operations");
printf("\n \t 1.PUSH");
printf("\n \t 2.POP");
printf("\n \t 3.DISPLAY");
printf("\n \t 4.EXIT");
printf("\n enter ur choice");
scanf("%d",&ch);
switch(ch)
{
case 1:printf("\n enter an element to push ");
 scanf("%d",&a);
push(a);
break;
case 2:
b=pop();
if(b!=-1)
printf("\n deleted element is %d ",b);
 break;
case 3:
display();
break;
 case 4:
 exit(0);
```

```
default:printf("\n ur choice is wrong");
}
}
return 0;
```

Application of stacks:

The following are three important applications of stack:

1. Recursion: ability to call itself.
2. Classical: deals with compilation of infix expression into object code.
3. Stack machine: certain computers perform stack operations at hardware level, and these operations enable insertions to and deletions from a stack to be made very rapidly.

The first application of the stack involves determining whether parentheses are balanced properly in algebraic expressions.

Let us concentrate on second application. Consider two numbers a and b.

i.e. c = a + b

Infix notation: whenever the operator is placed between the two operands then that type of expressions are called as Infix expression.

Examples:

1. a-b
2. a + b - c
3. (a + b)+(c / d)

Prefix notation: where the operator precedes the two operands. It is also called as POLISH NOTATION.

Examples:

1. –ab
2. -+abc
3. ++ab/cd

Postfix notation: where the operator follows the two operands. It is also called as REVERSE POLISH NOTATION.

1. ab-
2. abc-+

ab+cd/+

In computers, usually the expression written in infix notation is evaluated by first converting into postfix expressions and then we evaluate the postfix expression. Stacks are useful for the conversion and evaluations steps.

Conversion from Infix to Postfix Expression:

Algorithm:

1. Scan the input expression from right to left taking one symbol at a time. Till end of the given infix string, Goto step 5.
2. If an operand is encountered, push into the stack Goto step1.
3. if a left parenthesis is encountered, push it into the stack Goto step1.
4. If a right parenthesis is encountered, pop the stack and append the operator to output. Until the left parenthesis is encountered. Discard both parenthesis is encountered.
5. If an operator is encountered (current operator) co, then the top most element from the stack is retrieved (STO) and the priorities of the both the operators (CO & STO) are compared.

- If the STO (stack top operator) has higher or same priority as the current operator (CO) scanned, then STO is added to the output & CO is pushed into the stack.
- Else STO is pushed back to the stack then the current operator (CO) is also added to the stack.

Stack application- infix to postfix conversion:

```
#include<stdio.h>
#include<conio.h>
#define size 20 char s[size];
int top=-1;
void push(char ch)
{
top++;
s[top]=ch;
}
char pop()
{
char ch; ch=s[top];
top--;
return ch;
}
char topmost()
```

```
{
return s[top];
}
int icp(char ch)
{
switch(ch)
{
case'+':
case'-':
return 1;

case'*':
case'/':



case'(':

return 2;


return 5;

case'^':
return 4;
}
return 0;
}
int isp(char ch)
{
switch(ch)
{
case'+':
case'-':
return 1; case'*':
case'/':



case'(':

return 2;


return 0;

case'^':
return 4; case'#':
return -1;
}
return 0;
```

```
}
void convert(char a[],char b[])
{
int i,j; push('#'); j=0;

for(i=0;a[i]!='\o';i++)
{
if(a[i]>='a' && a[i]<='z')
{
b[j]=a[i]; j++;
}
else if(a[i]==')')
{
while(topmost()!='(')
{
b[j]=pop(); j++;
}
pop();
}
else if(icp(a[i])>isp(topmost()))
{
push(a[i]);
}
else
{
while(icp(a[i])<=isp(topmost()))
{
b[j]=pop(); j++;
}
push(a[i]);
}
}
while(topmost()!='#')
{
b[j]=pop(); j++;

} b[j]='\0';
}
int main()
{
char a[20],b[20];
printf("\nEnter the infix notation:");
 scanf("%s",a);
convert(a,b);
printf("\nPstfix Notation \n");
 printf("%s",b);
return 0;
}
```

Evaluation of Postfix Expression:-

Algorithm:

1. Scan the input expression from right to left taking one symbol at a time.
2. If an operand is encountered, push into the stack; else pop the two operand from the stack & apply the indicted operation & push the result into the stack.
3. Repeat the steps 1 and 2 until end of the expression.
4. Pop the Top element of the stack, which is the result of the expression.

Example: Evaluation of postfix expression

1. abc/d*+ for a+(b/c)*d infix exp.

Symbol	Stack	
A	5	where
B	5 4	a=5, b=4, c=2
C	5 4 2	
/	5 2	
D	5 2 2	
*	5 4	
+	9	

Result: 9

NOTE: whenever we see a oprator in the expression we have to pop the two operands & perform that operation of operator on it. Result is placed in stack

```
Postfix evaluation
#define size 10
int s[size];
int top=-1;
void push(int n)
{
top++;
s[top]=n;
}
int pop()
{
return s
[top--];
}
int topmost()
{
return s[top];
}
int eval(char a[])
```

```
{
int i,x,y,dx;
 for(i=0;a[i]!='\0';i++)
{
if(a[i]>='0' && a[i]<='9')
{
push(a[i]-'0');
}
else
{
y=pop();

x=pop();
switch(a[i])
{
case '+':
push(x+y);
break;
case '-':
push(x-y); break;
case '*':
push(x*y); break;
case '^':
dx=pow(x,y);
push(dx); break;
case '/':
push(x/y); break;
}
}
}
return topmost();
}
int main()
{
char a[20];
 int rs;
printf("\nEnter the postfix Notation:");
scanf("%s",&a);
rs=eval(a);
printf("\nResult is: %d",rs);
 return 0;

}
```

QUEUES:

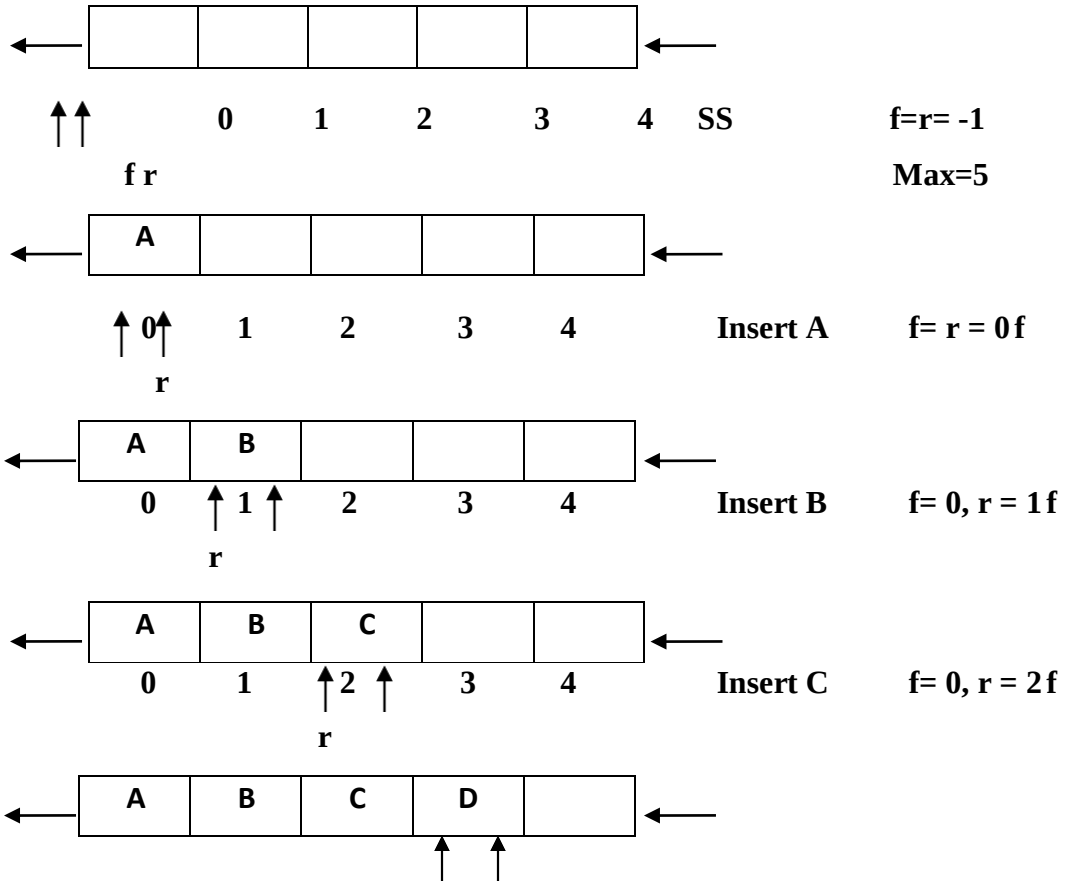
Queue is an ordered collection of data, such that the elements or data is inserted at one end and deleted at another end.

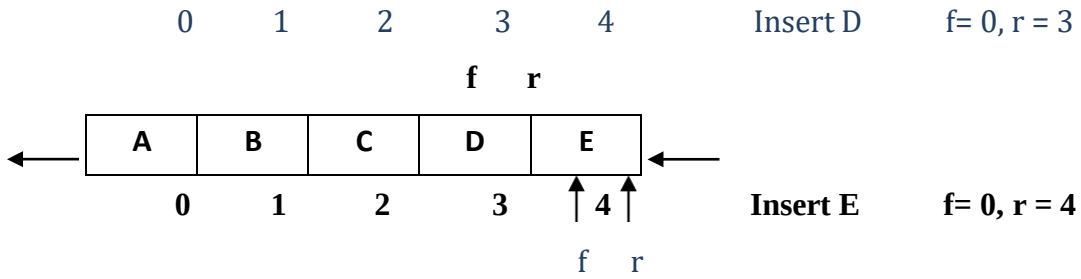
- The elements are inserted at one end called Rear and at another end called Front.
- Queue is processed on a FIFO STRUCTURE (FIRST IN FIRST OUT).
- It works on that the elements inserted first, in the element to be deleted.
- Queue is associated with the two operations insert and delete.

Insert: To insert an element into the queue by checking its overflow condition an element is inserted, at the rear end we can insert, then rear is incremented. If rear equal to maximum size of queue, then queue is said to be overflow condition.

Delete: To delete an element from the queue by checking its underflow condition whenever an element is deleted, front is incremented. If front is equal to zero or front is equal to rear, then queue is said to be in underflow condition.

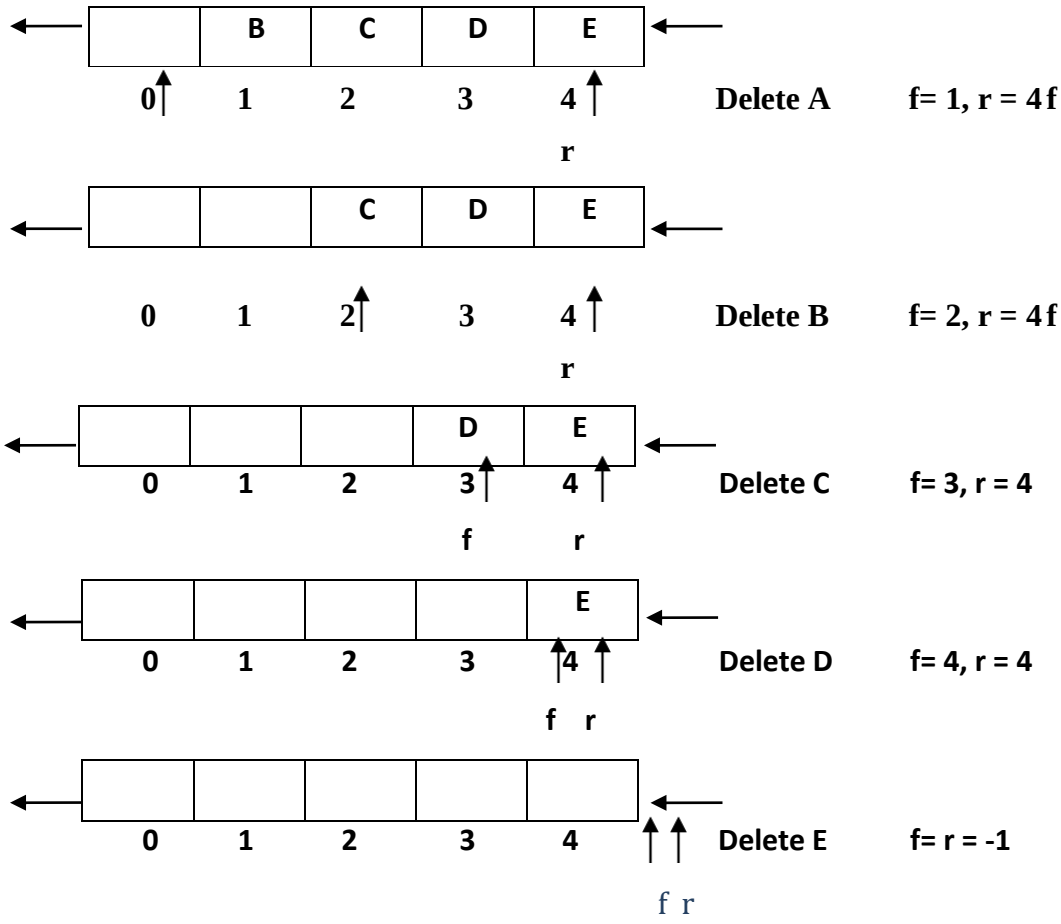
Insertion:





Note: Again if we want to insert it will display queue is overflow

Deletion



Note: whenever we are inserting the element for the first time into the queue, then set f=0.

Algorithm for inserting an element into a queue

1. Initialize front=-1 and rear=-1
2. if rear==MAX-1

   Print "Queue is overflow" and return Else

   Increment rear Set

   rear= rear+1

3. Q[rear] =ITEM
4. if (front=-1) then set front=0
5. Exit

Algorithm for deleting an element into a queue

1. Check for Queue under flow

   If(rear==-1)

   Print Queue is under flow and return Else

   ITEM = Q[front]// Removes an element from queue
2. Find new value of front If(front=rear)//checking

   for empty queue

   Set front=-1 and rear=-1//Reinitialize pointers Else
3. Front = front +1

Exit

Queues using Arrays

```
#include<process.h>
 #define size 5
int q[size];
int front=0;
 int rear=-1;
void insert(int n)
{
if(rear==size-1)
{
printf("\nQueue overflow");
}
else
{
rear++; q[rear]=n;
printf("\nelement inserted");
}
}
int del()
{
int dx;

if(rear==-1)
{
printf("\nQueue underflow");
return-1;
}
else if(front==rear)
{
dx=q[front];
front=0;
```

```
rear=-1;
 return dx;
}
else
{
dx=q[front];
front++;
return dx;
}
}
void display()
{
int i;
 if(rear==-1)
{
printf("\nQueue is empty");
}
else
{
printf("\nElements of queue are \n");
for(i=front;i<=rear;i++)
{
printf("%5d",q[i]);

}
int main()
{
int n,dx,ch;
 while(1)
{
printf("\nOperations on stacks");
printf("\n1.Insert");
printf("\n2.Delete");
printf("\n3.Display");
printf("\n4.Exit");
 printf("\nEnter ur choice");
 scanf("%d",&ch);
switch(ch)
{
case 1:

printf("\nEnter the element to insert"); scanf("%d",&n);
insert(n); break;

case 2:

dx=del();
if(dx!=-1)
printf("\nDeleted element is:%d",dx); break;

case 3:
```

```
display(); break;

case 4: exit(0);
default: printf("\nInvalid Choice");
}
return 0;
```

Linked list

List: A list refers to a set of items organized sequentially.

Advantages:

- Linked list are dynamic Data Structures. i.e. they can grow or shrink during the execution of a program.
- Efficient memory utilization. Here, memory is pre-allocated. Memory is allocated whenever it is required. And it is deallocated(removed) when it is no longer needed.
- Insertion and deletion are easier. Linked lists provide flexibility in inserting a data item provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages:

- More memory: if the number of fields is more, then more memory space is needed.
- Access to an arbitrary data item is little bit cumbersome and also time –consuming.

Advantages:

- A linked list is a dynamic data structure which means that the memory to hold the data is created at runtime (dynamic), and stored in a data structure.
- A linked list is a linear data structure. i.e. it is a collection of similar data elements called nodes. The individual elements are stored at different places in memory but still bounded together by means of pointer.
- Each node in the linked list consists of two fields, information fields and link part information fields contains data item and the link part contains the address of the most item in the list
- The last node of the linked list contains a special link part called NULL pointer) i.e. any invalid address).

Disadvantages:

- It consumes more space because every node requires an additional pointer to store address of the next node.
- Searching a particular element in list is difficult and also time consuming.

Application of linked lists

- Linked lists are used to implement stack, queue, trees and graphs.
- Linked lists are used to represent and manipulate polynomial P(x) = $a_0x^n+a_1x^{n-1}+$…….
- Represent very large numbers and operations of the large number such as addition, multiplication, division.

Single Linked List

```
#include<stdio.h>
#include<conio.h>
 #include<alloc.h>
 struct node
{
int data;
struct node *next;
};


struct node *root=NULL;
 void create()
{
struct node *new1,*last; int n;
while(1)
{
printf("enter the data or -1 to stop");
 scanf("%d",&n);
if(n==-1) break;
new1=(struct node*)malloc(sizeof(struct node));
 new1->data=n;
new1->next=NULL;
if(root==NULL)

{
root=last=new1;
}
else
{
last->next=new1;
 last=new1;
}
```

```
}
}
void display()
{
struct node *temp;
if(root==NULL)
{
printf("list is empty");
}
else
{
printf("\n elements of list are");
 temp=root; while(temp!=NULL)
{
printf("%d",temp->data);
 temp=temp->next;
 if(temp!=NULL)
printf("--->");
}
}
}
int length()
{
int len=0;
struct node *temp;
temp=root;

while(temp!=NULL)
{
len++;
temp=temp->next;
}
return len;
// getch();
}
void insert(int pos,int n)
{
struct node *temp,*new1; int i;
if(pos<1||pos>length()+1)
{
printf("\n invalid position");
}
else
{
new1=(struct node*)malloc(sizeof(struct node));
new1->data=n;
new1->next=NULL;
if(pos==1)
{
```

```
new1->next=root;
 root=new1;
}
else
{
temp=root; for(i=1;i<pos-1;i++)
temp=temp->next;
new1->next=temp->next;
 temp->next=new1;
}
}
}

int del(int pos)
{
struct node *p,*temp;
 int dx,i;
 if(pos<1||pos>length())
{
printf("\n invalid choice");
 return -1;
}
else if(pos==1)
{
temp=root; dx=root->data;
 root=root->next;
free(temp);
 return dx;
}
else
{
temp=root; for(i=1;i<pos-1;i++)
temp=temp->next; p=temp->next;
temp->next=p->next; dx=p->data;
free(p); return dx;
}
}
int main()
{
int dx,pos,n,ch;
create();
display();
while(1)
{

// clrscr();
printf("\n operations on linked lists");
printf("\n1.Insert");
 printf("\n2.Delete");
 printf("\n3.Display");
printf("\n4.Length");
```

```
printf("\n5.Exit");
printf("\nEnter ur choice:");
 scanf("%d",&ch);
switch (ch)
{
case 1: printf("enter the position to insert");
scanf("%d",&pos);
printf("enter the element to be inserted:");
 scanf("%d",&n);
insert(pos,n);
break;
case 2: printf("enter the position to delete");
scanf("%d",&pos);
dx=del(pos);
if(dx!=-1)
printf("\n Deleted element=%d",dx);
 break;
case 3: display();
break;
case 4: dx=length();
printf("\n no of nodes=%d",dx);
 break;
case 5: exit(0);
default: printf("\n invalid choice");
}// getch();
}
return 0;
}
```

Stacks using linked list

```
#include<stdio.h>
 #include<conio.h>
 typedef struct node
{
int data;
struct node *link;
}list;
list *top=NULL;
void push(int x)
{
list *new1;
new1=(list *)malloc(sizeof(list));
new1->data=x;
new1->link=top;
 top=new1;
}
int pop()
{
list *temp; int y;
if(top==NULL)
{
```

```
printf("\n stack underflow");
 y=-1;
}
else
{
temp=top;
top=top->link;
y=temp->data;
 free(temp);
}
return y;
}
void display()
{

list *temp; if(top==NULL)
printf("\n satck underflow");
 else
{
temp=top;
while(temp!=NULL)
{
// printf("\n stack elements are: ");
 printf("%4d",temp->data);
 temp=temp->link;        } }         }
int main()
{
int ch,a,b; while(1)
{
printf("\n stack operations");
 printf("\n \t 1.PUSH");
printf("\n \t 2.POP");
printf("\n \t 3.DISPLAY");
printf("\n \t 4.EXIT");
 printf("\n enter ur choice");
 scanf("%d",&ch);
 switch(ch)
{
case 1:printf("\n enter an element to push ");
scanf("%d",&a);
push(a); break;
case 2:b=pop();
if(b!=-1)
printf("\n deleted element is %d ",b);
break;
case 3:display();
break;

case 4: exit(0);
default:printf("\n ur choice is wrong");
}
```

```
}
return 0;
}
```

Queues using Linked Lists

```
#include<alloc.h>
struct node
{
int data;
struct node *next;
};
struct node *front=NULL;
 struct node *rear=NULL;
 void insert(int n)
{
struct node *new1;
new1=(struct node *)malloc(sizeof(struct node));
new1->data=n;
new1->next=NULL;
if(front==NULL)
{
front=rear=new1;
}
else
{
rear->next=new1;
 rear=new1;
}
}
int del()
{
int dx;
struct node *temp;
 if(front==NULL)

{
printf("\nQueue underflow");
return -1;
}
else
{
dx=front->data; temp=front;
 front=front->next;
free(temp);
return dx;
}
}
void display()
{
```

```
struct node *temp;
 if(front==NULL)
{
printf("\nWQueue is empty");
}
else
{
printf("\nElements of Queue are\n");
 temp=front;
while(temp!=NULL)
{
printf("%5d",temp->data);
 temp=temp->next;
}
}
}
int main()
{
int n,dx,ch; while(1)
{

clrscr();
printf("\nOperations on stacks");
printf("\n1.Insert");
 printf("\n2.Delete");
printf("\n3.Display");
printf("\n4.Exit");
printf("\nEnter ur choice");
 scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nEnter the element to insert");
scanf("%d",&n);
insert(n); break;
case 2: dx=del();
if(dx!=-1)
printf("\nDeleted element is:%d",dx); break;

case 3:



case 4:


display();
 break;

exit(0);

default:
```

```
printf("\nInvalid Choice");
}
return 0;
}
}
```

Case Studies

Case 1: Student Record Management System

The main features of this project include basic file handling operations; you will learn how to add, list, modify and delete data to/from file. The source code is relatively short, so thoroughly go through the mini project, and try to analyze how things such as functions, pointers, files, and arrays are implemented.

Currently, listed below are the only features that make up this project, but you can add new features as you like to make this project a better one!

- Add record
- List record
- Modify record
- Delete record

Case 2: Library Management System

This project has 2 modules.

1. Section for a librarian
2. Section for a student

A librarian can add, search, edit and delete books. This section is password protected. That means you need administrative credentials to log in as a librarian.

A student can search for the book and check the status of the book if it is available.Here is list of features that you can add to the project.

1. You can create a structure for a student that uniquely identify each student. When a student borrows a book from the library, you link his ID to Book ID so that librarian can find how burrowed particular book.
2. You can create a feature to bulk import the books from CSV file.
3. You can add REGEX to search so that a book can be searched using ID, title, author or any of the field.
4. You can add the student login section.