COMPUTER PROGRAMMING AND DATA STRUCTURES

CONTENTS

Important Questions 201

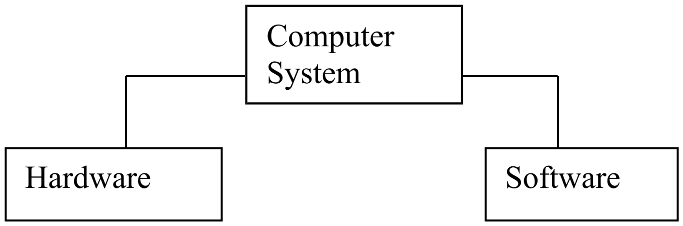
COMPUTER PROGRAMMING AND DATA STRUCTURES

<u>UNIT - I</u>

Introduction to Computers:

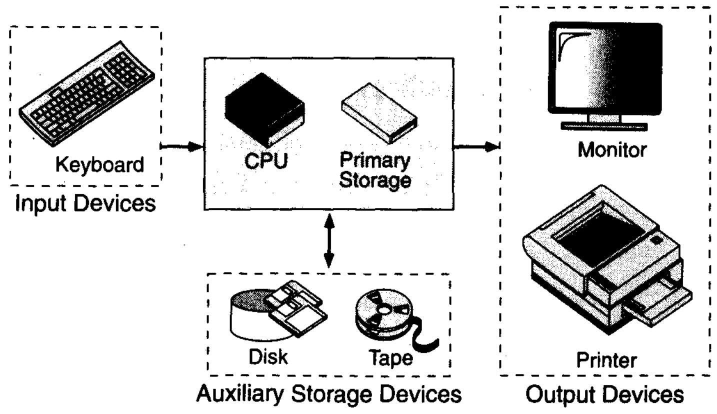
Computer Systems:

A computer is a system made of two major 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.



Computer Hardware

The hardware component of the computer system consists of five parts: input devices, central processing unit (CPU) ,primary storage, output devices, and auxiliary storage devices.



Basic Hardware Components

The **input device** is usually a keyboard where programs and data are entered into the computers. Examples of 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 instructions such as arithmetic calculations,comparisons among data, and movement of data inside the system. Today's computers may have one ,two, or more CPUs .**Primary storage** ,also known as main memory, is a place where the programs and data are stored temporarily during processing. The data in primary storage are erased when we turn off a personal computer or when we log off from a time-sharing system.

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 it is printed on the printer, we say we have a hard copy.

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

Computer Software

Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.

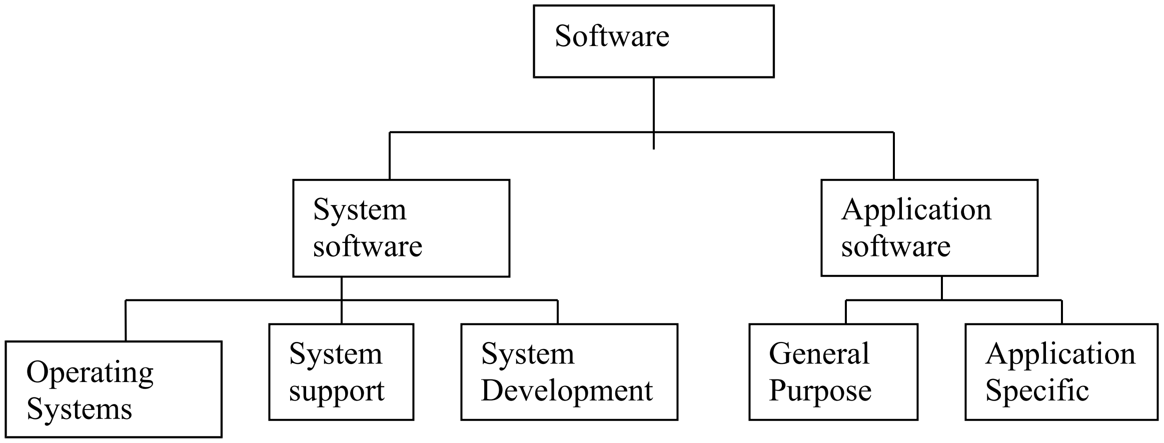


Fig: Types of software

System Software:

System software consists of programs that manage 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 interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

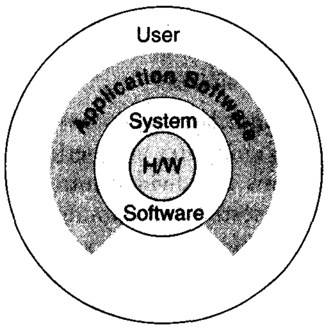
The last system software category ,**system development software**, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

Application software

Application software is broken in to 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 of general purpose software include word processors ,database management systems ,and computer aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

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 of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relation ship between system and application software is shown in fig-2.In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system,the typical user uses some form of application software. The application software in turn interacts with the operating system ,which is apart of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.



Relationship Between System and Application Software

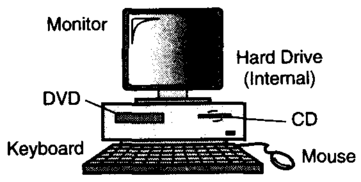
Computing Environments:

In the early days of computers, there was only one environment: the main frame computer hidden in a central computing department. With the advent of mini computers and personal computers, 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. The first computer on a chip was the Intel 4004 and was the grandparent many times removed of Intel's current system.

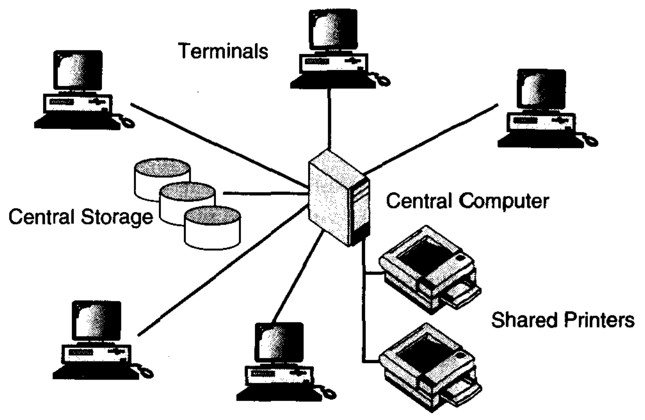
If we are using a personal computer, all of the computer hardware components are tied together in our personal computer(PC).



Personal Computing Environment

Time-Sharing Environment

Employees in large companies often work in what is known as a **time-sharing environment.** In the times-sharing environment, many users are connected to one or more computers. These computers may be minicomputers or central mainframes. The terminals they use are often nonprogrammable, although today we see more and more microcomputers being used to simulate terminals. Also, in the time-sharing environment, the output devices and auxiliary storage devices are shared by all of the users. A typical college lab in which a minicomputer is shared is shared by many students is shown in figure :

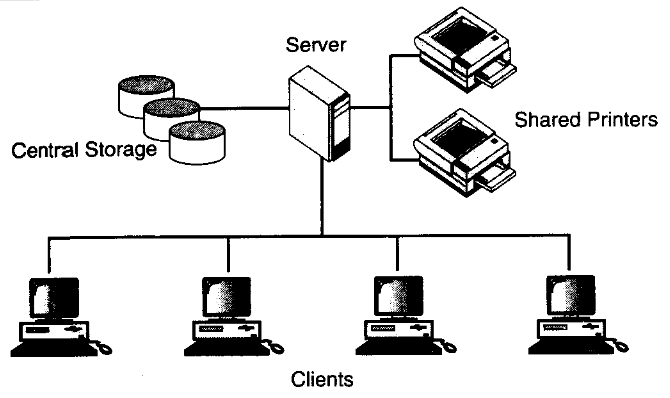


Time-sharing Environment

In the time-sharing environment ,all computing must be done by the central computer. 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 users' computers. The users are given personal computers or work stations so that some of the computation responsibility can be moved from the central computer and assigned to the workstations. In the client-server environment, the users' micro computers or workstations are called the **client**. The central computer, which may be a powerful microcomputer, minicomputer, or central mainframe system, is known as the **server.** Because the work is now shared between the users' computers and the central computer, response time and monitor display are faster and the users are more productive.

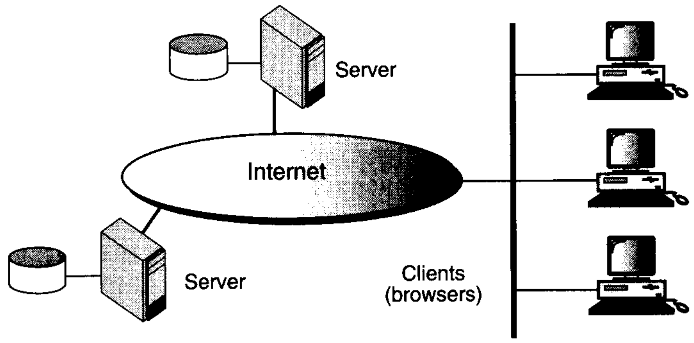


The Client/Server Environment

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. For example eBay uses several computers to provide its auction services. This environment provides a reliable, scalable, and highly available network.

Fig: Distributed Computing



Distributed Computing

Computer Languages:

To write a program for a computer, we must use a **computer language.** Over the years computer languages have evolved from machine languages to natural languages.

1940's	Machine level Languages
1950's	Symbolic Languages
1960's	High-Level Languages

Machine Languages

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language, which is made of streams of 0's and 1's.
Instructions in machine language must be in streams of 0's and 1's because the internal circuits of a computer are made of switches transistors and other electronic devices that can be in one of two states: off or on. The off state is represented by 0 , the on state is represented by 1.
The only language understood by computer hardware is machine language.

Symbolic Languages:

In early 1950's Admiral Grace Hopper, A mathematician and naval officer developed the concept of a special computer program that would convert programs into machine language. The early programming languages simply mirror to the machine languages using symbols of mnemonics to represent the various machine language instructions because they used symbols, these languages were known as symbolic languages. Computer does not understand symbolic language it must be translated to the machine language. A special program called assembler translates symbolic code into machine language. Because symbolic languages had to be assembled into machine language they soon became known as assembly languages.
Symbolic language uses symbols or mnemonics to represent the various ,machine language instructions.

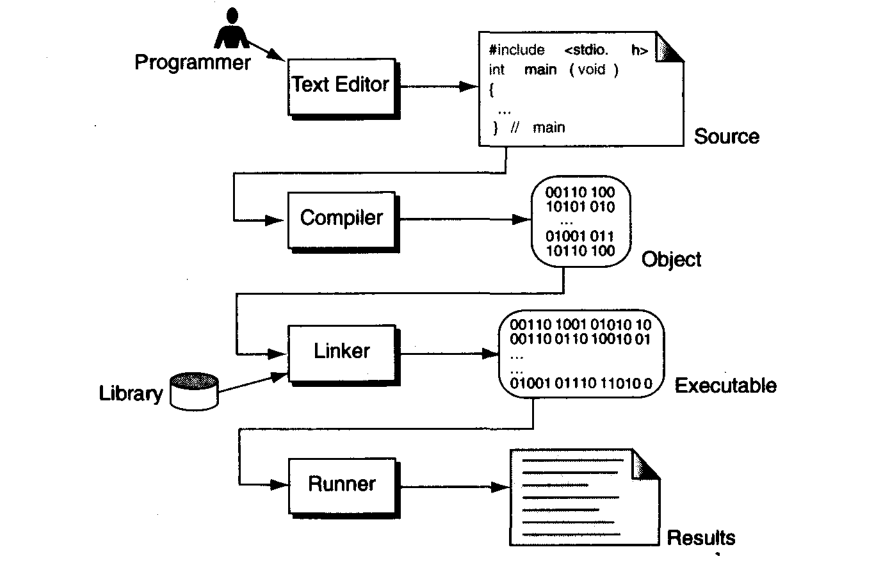
High Level Languages:

Symbolic languages greatly improved programming effificiency; they still required programmers to concentrate on the hardware that they were using. Working with symbolic languages was also very tedious because each machine instruction has to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of high-level language.
High level languages are portable to many different computers, allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer. High-level languages are designed to relieve the programmer from the details of the assembly language. High level languages share one thing with symbolic languages, They must be converted into machine language. The process of converting them is known as compilation.

The first widely used high-level languages, FORTRAN (FORmula TRANslation)was created by John Backus and an IBM team in 1957;it is still widely used today in scientific and engineering applications. After FORTRAN was COBOL(Common Business-Oriented Language). Admiral Hopper was played a key role in the development of the COBOL Business language.
C is a high-level language used for system software and new application code.

Creating and Running Programs:

Computer hardware understands a program only if it is coded in its machine language. It is the job of the programmer to write and test the program .There are four steps in this process:1.Writing and Editing the program2.Compiliing the program 3.Linking the program with the required library modules 4.Executing the program.



Building a C Program

Writing and Editing Programs

The software used to write programs is known as a **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, create reports, or write programs. The main difference between text processing and program writing is that programs are written using lines of code, while most text processing is done with character and lines.

Text editor is a generalized word processor, but it is more often a special editor included with the compiler. Some of the features of the editor are search commands to locate and replace statements, copy and paste commands to copy or move statements from one part of a program to another, and formatting commands that allow us to set tabs to align statements.
After completing 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 two separate programs. the **preprocessor** and the **translator.**

The preprocessor reads the source code and prepares it for the translator. While preparing the code ,it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries, make substitutions in the code ,and in other ways prepare the code for translation into 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. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in machine language. The output of the compiler is machine language code, but it is not ready to run; that is ,it is not executable because it does not have the required C and other functions included.

Linking Programs:

A C program is made up of many functions. We write some of these functions, and they are a part of our source program. There are other functions, such as input/output processes and, mathematical library functions, that exist elsewhere and must be attached to our program. The linker assembles all of these functions, ours and systems into our final executable program.

Executing Programs:

Once 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.

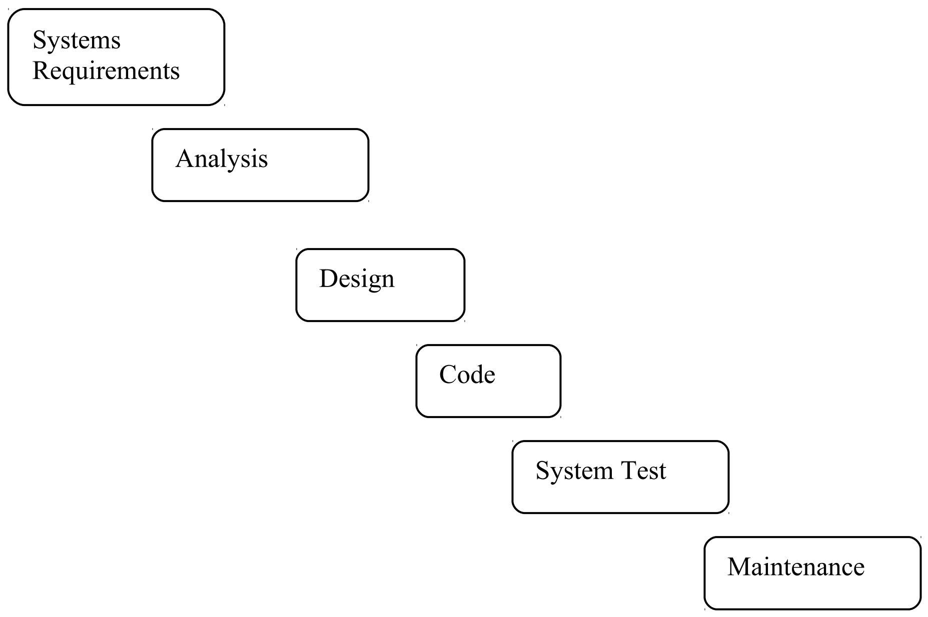
In a typical program execution, the reads data for processing ,either from the user or from a file. After the program processes the data, it prepares the output. at output can be to the user's monitor or to a file. When the program has finished its job, it tells the operating system ,which then removes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

System Development Method

A **software development process** is a structure imposed on the development of a software product. This critical process determines the overall quality and success of our program. If we carefully design each program using good structured development techniques, programs will be efficient, error-free, and easy to maintain.

System Development Life Cycle

Today's large-scale modern programming projects are built using a series of interrelates phases commonly referred to as the system development cycle. Although the exact number and names of the phases differ depending on the environment there is general agreement as to the steps that must be followed. One very popular development life cycle developed, this modal consists of between 5 and 7 phases.



The water fall modal starts with systems requirements in this phase the systems analyst defines requirements that specify what the proposed system is to accomplish. The requirements are usually stated in terms that the user understands. The analysis phase looks at different alternatives from a systems point of view while the design phase determined how the system will be built. In the design phase the functions of the individual programs that will make up the system are determined and the design of the files and / or the databases is completed. Finally in the 4th phase code, we write the programs. After the programs have been written and tested to the programmer's satisfaction, the project proceeds to the system test. All of the programs are tested together to make sure of the system works as a whole. The final phase maintenance keeps the system working once it has been put into production.
Although the implication of the water falls approach is that the phases flow in a continuous stream from the first to the last, this is not really the case. As each phase is developed, errors and omissions will often be found in the previous work. When this happens it is necessary to go back to the previous phase to rework it for consistency and to analyze the impact caused by the changes.

ALGORITHM /PSEUDOCODE:

PSEUDOCODE:

Pseudo code is an artificial and informal language that helps programmers develop algorithams.pseudocode is similar to everyday English, it is convenient and user friendly although it is not an actual computer programming language. Psuedocode programs are not actually executed on computer rather they merely help the programmer "think out" a program before attempting to write it in a programming language such as C.

ALGORITHM:

Algorithms was developed by an Arab mathematician. It is chalked out step-by-step approach to solve a given problem. It is represented in an English like language and has some mathematical symbols like ->, >, <, = etc. To solve a given problem or to write a program you approach towards solution of the problem in a systematic, disciplined, non-adhoc, step-by-step way is called Algorithmic approach. Algorithm is a penned strategy(to write) to find a solution.

Example: Algorithm/pseudo code to add two numbers

```
Step 1: Start
Step 2:Read the two numbers in to a,b
Step 3: c=a+b
```

Step 4: write/print c
Step 5: Stop.

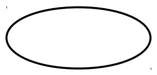
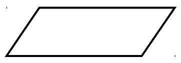
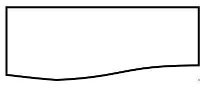
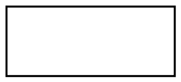
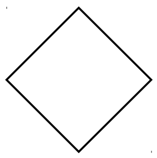
FLOW CHART :

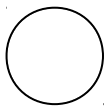
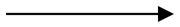
A Flow chart is a Graphical representation of an Algorithm or a portion of an Algorithm. Flow charts are drawn using certain special purpose symbols such as Rectangles, Diamonds, Ovals and small circles. These symbols are connected by arrows called flow lines.

(or)

The diagrammatic representation of way to solve the given problem is called flow chart.

The following are the most common symbols used in Drawing flowcharts:

Oval		Terminal	start/stop/begin/end.
Parallelogram		Input/output	Making data available For processing(input) or recording of the process information(output).
Document		Print Out	show data output in the form of document.
Rectangle		Process	Any processing to be Done .A process changes or moves data.An assignment operation.
Diamond		Decision	Decision or switching

			type of operations.
Circle		Connector	Used to connect Different parts of flowchart.
Arrow		Flow	Joins two symbols and also represents flow of execution.

PROGRAM DEVELOPMENT STEPS :

Program Development is a multistep process that requires that we understand the problem, develop a solution, write the program, and then test it. When we are given the assignment to develop a program, we will be given a program requirements statement and the design of any program interfaces. We should also receive an overview of the complete project so that we will take the inputs we are given and convert them to the outputs that have been specified. This is known as program design.

Understand the Problem

The first step in solving any problem is to understand it. By reading the requirements statements carefully, we fully understand it, we review our understanding with the user and the systems analyst to know the exact purpose.

Develop the solution

Once we fully understand the problem we need to develop our solution. Three tools will help in this task. 1. Structure chart, 2.Psuedocode &3.Flowcharts. Generally we will use structure chart and either flowchart or Pseudo code

The structure chart is used to design the whole program .Pseudo code and flowcharts are used to design the individual parts of the program.

Structure chart: A structure chart, also known as hierarchy chart, shows the functional flow through our program. The structure chart shows how we are going to break our

program into logical steps each step will be a separate module. The structure chart shows the interaction between all the parts (modules) of our program.
We can use flowchart or pseudo code to complete the design of your program will depend on experience and difficulty of the program your designing.

Write the program

When we write a program, we start with the top box on the structure chart and work our way to the bottom. This is known as top-down implementation. We will write the programs by using structure chart and flowchart or pseudo code.

Test the Program

Program testing can be a very tedious and time- consuming part of program development. As the programmer we are responsible for completely testing our program. In large-development projects test engineers are responsible for testing to make sure all the programs work together.
There are 2 types of testing.

1. **Black box testing:** This is done by the system test engineer and the user. Black box testing is the programs are tested without knowing what is inside it, with out knowing how it works. Black box test plans are developed by looking only the requirements statement. The test engineer uses these requirements to develop test plans.
2. **White box testing:** This is the responsibility of the programmer. White box testing assumes that the tester knows everything about the program.

Except for the simplest program, one set of test data will not completely validate a program.

IMPORTANT QUESTIONS UNIT-I

1. What is a flowchart? Explain the different symbols used in a flowchart.

2.(a) Define an Algorithm?

(b) What is the use of Flowchart?

(c) What are the different steps followed in the program development?

UNIT-II

INTRODUCTION TO ‘C’ LANGUAGE:

C language facilitates a very efficient approach to the development and implementation of computer programs. The History of C started in 1972 at the Bell Laboratories, USA where Dennis M. Ritchie proposed this language. In 1983 the American National Standards Institute (ANSI) established committee whose goal was to produce “an unambiguous and machine independent definition of the language C “ while still retaining it’s spirit .

C is the programming language most frequently associated with UNIX. Since the 1970s, the bulk of the UNIX operating system and its applications have been written in C. Because the C language does not directly rely on any specific hardware architecture, UNIX was one of the first portable operating systems. In other words, the majority of the code that makes up UNIX does not know and does not care which computer it is actually running on. Machine-specific features are isolated in a few modules within the UNIX kernel, which makes it easy for you to modify them when you are porting to a different hardware architecture.

C was first designed by Dennis Ritchie for use with UNIX on DEC PDP-11 computers. The language evolved from Martin Richard's BCPL, and one of its earlier forms was the B language, which was written by Ken Thompson for the DEC PDP-7. The first book on

C was *The C Programming Language* by Brian Kernighan and Dennis Ritchie, published in 1978.

In 1983, the American National Standards Institute (ANSI) established a committee to standardize the definition of C. The resulting standard is known as *ANSI C,* and it is the recognized standard for the language, grammar, and a core set of libraries. The syntax is slightly different from the original C language, which is frequently called K&R for Kernighan and Ritchie. There is also an ISO (International Standards Organization) standard that is very similar to the ANSI standard.

It appears that there will be yet another ANSI C standard officially dated 1999 or in the early 2000 years; it is currently known as "C9X."

BASIC STRUCTURE OF C LANGUAGE :

The program written in C language follows this basic structure. The sequence of sections should be as they are in the basic structure. A C program should have one or more sections but the sequence of sections is to be followed.

```
1. Documentation section
2. Linking section
3. Definition section
4. Global declaration section
5. Main function section
        {
         Declaration section
         Executable section
        }
6. Sub program or function section
```

1. DOCUMENTATION SECTION : comes first and is used to document the use of logic or reasons in your program. It can be used to write the program's objective, developer and logic details. The documentation is done in C language with /* and */ . Whatever is written between these two are called comments.

2. LINKING SECTION : This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.

e.g. #include <stdio.h>

3. DEFINITION SECTION : It is used to declare some constants and assign them some value.

```
e.g.   #define MAX 25
```

Here #define is a compiler directive which tells the compiler whenever MAX is found in the program replace it with 25.

4. GLOBAL DECLARATION SECTION : Here the variables which are used through out the program (including main and other functions) are declared so as to make them global(i.e accessible to all parts of program)
e.g. int i; (before main())

5. MAIN FUNCTION SECTION : It tells the compiler where to start the execution from

```
main()
{
        point from execution starts
}
```

main function has two sections

1. declaration section : In this the variables and their data types are declared.
2. Executable section : This has the part of program which actually performs the task we need.

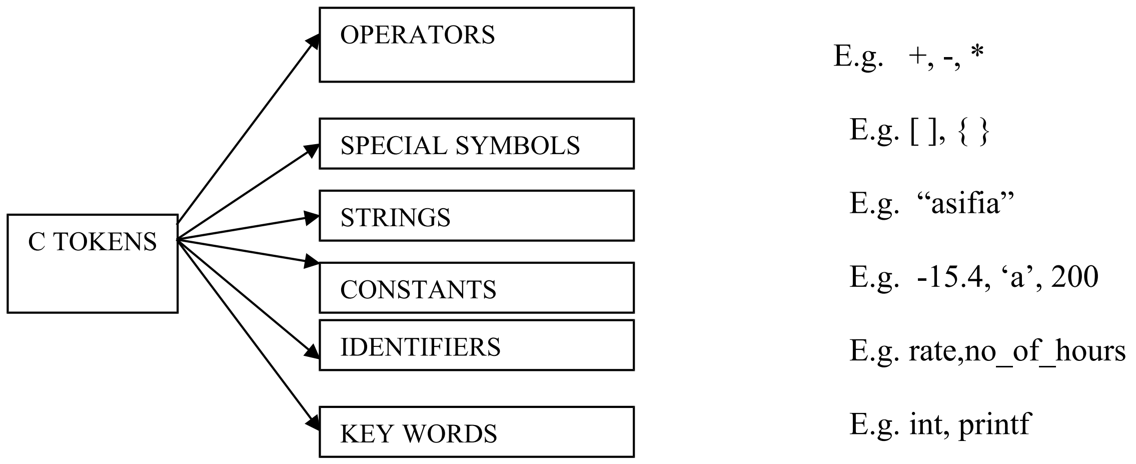
6. SUB PROGRAM OR FUNCTION SECTION : This has all the sub programs or the functions which our program needs.

SIMPLE 'C' PROGRAM:

```
/* simple program in c */
#include<stdio.h>
main()
{
printf("welcome to c programming");
} /*  End of main */
```

C-TOKENS :

Tokens are individual words and punctuations marks in English language sentence. The smallest individual units are known as C tokens.



A C program can be divided into these tokens. A C program contains minimum 3 c tokens no matter what the size of the program is.

DATA TYPES :

To represent different types of data in C program we need different data types. A data type is essential to identify the storage representation and the type of operations that can be performed on that data. C supports four different classes of data types namely

1. Basic Data types
2. Derives data types
3. User defined data types
4. Pointer data types

BASIC DATA TYPES:

All arithmetic operations such as Addition , subtraction etc are possible on basic data types.

E.g.: int a,b;
Char c;

The following table shows the Storage size and Range of basic data types:

TYPE	LENGTH	RANGE
Unsigned char	8 bits	0 to 255
Char	8 bits	-128 to 127
Short int	16 bits	-32768 to 32767
Unsigned int	32 bits	0 to 4,294,967,295
Int	32 bits	-2,147,483,648 to 2,147,483,648
Unsigned long	32 bits	0 to 4,294,967,295
Enum	16 bits	-2,147,483,648 to 2,147,483,648
Long	32 bits	-2,147,483,648 to 2,147,483,648
Float	32 bits	3.4*10E-38 to 3.4*10E38
Double	64 bits	1.7*10E-308 to 1.7*10E308
Long double	80 bits	3.4*10E-4932 to 1.1*10E4932

DERIVED DATA TYPES:

Derived datatypes are used in ‘C’ to store a set of data values. Arrays and Structures are examples for derived data types.

```
Ex:   int a[10];
      Char name[20];
```

USER DEFINED DATATYPES:

C Provides a facility called typedef for creating new data type names defined by the user. For Example ,the declaration ,

typedef int Integer;

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

POINTER DATA TYPES:

Pointer data type is necessary to store the address of a variable.

VARIABLES :

A quantity that can vary during the execution of a program is known as a variable. To identify a quantity we name the variable for example if we are calculating a sum of two numbers we will name the variable that will hold the value of sum of two numbers as 'sum'.

IDENTIFIERS :

Names of the variables and other program elements such as functions, array,etc,are known as identifiers.

There are few rules that govern the way variable are named(identifiers).

1. Identifiers can be named from the combination of A-Z, a-z, 0-9, _(Underscore).
2. The first alphabet of the identifier should be either an alphabet or an underscore. digit are not allowed.
3. It should not be a keyword.

Eg: name,ptr,sum

After naming a variable we need to declare it to compiler of what data type it is .
The format of declaring a variable is

```
Data-type  id1, id2,.....idn;
```

where data type could be float, int, char or any of the data types.

id1, id2, id3 are the names of variable we use. In case of single variable no commas are required.

```
eg      float a, b, c;
        int   e, f, grand total;
        char  present_or_absent;
```

ASSIGNING VALUES :

When we name and declare variables we need to assign value to the variable. In some cases we assign value to the variable directly like

```
a=10;
```

in our program.

In some cases we need to assign values to variable after the user has given input for that.

eg we ask user to enter any no and input it

```
/* write a program to show assigning of values to variables   */

#include<stdio.h>
```

```
main()
{
        int a;
        float b;
        printf("Enter any number\n");
        b=190.5;
        scanf("%d",&a);
        printf("user entered %d", a);
        printf("B's values is %f", b);
}
```

CONSTANTS :

A quantity that does not vary during the execution of a program is known as a constant supports two types of constants namely Numeric constants and character constants.

NUMERIC CONSTANTS:

1. Example for an integer constant is 786,-127
2. Long constant is written with a terminal ‘l’or ‘L’,for example 1234567899L is a Long constant.
3. Unsigned constants are written with a terminal ‘u’ or ‘U’,and the suffix ‘ul’ and ‘UL’ indicates unsigned long. for example 123456789u is a Unsigned constant and 1234567891ul is an unsigned long constant.
4. The advantage of declaring an unsigned constant is to increase the range of storage.
5. Floating point constants contain a decimal point or an exponent or both. For Eg : 123.4,1e-2,1.4E-4,etc.The suffixes f or F indicate a float constant while the absence of f or F indicate the double, l or L indicate long double.

CHARACTER CONSTANTS:

A character constant is written as one character with in single quotes such as ‘a’. The value of a character constant is the numerical value of the character in the machines character set. certain character constants can be represented by escape sequences like ‘\n’. These sequences look like two characters but represent only one.

The following are the some of the examples of escape sequences:

Escape sequence	**Description**
\a	Alert

\b	Backspace
\f	Form feed
\n	New Line
\r	Carriage return
\t	Horizontal Tab
\v	Vertical Tab

String constants or string literal is a sequence of zero or more characters surrounded by a double quote. Example , " I am a little boy". quotes are not a part of the string.

To distinguish between a character constant and a string that contains a single character ex: 'a' is not same as "a". 'a' is an integer used to produce the numeric value of letter a in the machine character set, while "a" is an array of characters containing one character and a '\0' as a string in C is an array of characters terminated by NULL.

There is one another kind of constant i.e Enumeration constant , it is a list of constant integer values.

Ex.: enum color { RED, Green, BLUE }

The first name in the enum has the value 0 and the next 1 and so on unless explicit values are specified.

If not all values specified , unspecified values continue the progression from the last specified value. For example

Enum months { JAN=1, FEB,MAR, …, DEC }

Where the value of FEB is 2 and MAR is 3 and so on.

Enumerations provide a convenient way to associate constant values with names.

KEYWORDS :

There are certain words, called keywords (reserved words) that have a predefined meaning in 'C' language. These keywords are only to be used for their intended purpose and not as identifiers.

The following table shows the standard 'C' keywords

Auto	Break	Case	Char	Const	Continue
Default	Do	Double	Else	Enum	Extern
Float	For	Goto	If	Int	Long
Register	Return	Short	Signed	Sizeof	Static
Struct	Switch	Typedef	Union	Unsigned	void
Volatile	While				

OPERATORS :

An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations. They form expressions.
C operators can be classified as

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment or Decrement operators
6. Conditional operator
7. Bit wise operators
8. Special operators

1. ARITHMETIC OPERATORS : All basic arithmetic operators are present in C.

operator	meaning
+	add
-	subtract
*	multiplication
/	division
%	modulo division(remainder)

An arithmetic operation involving only real operands(or integer operands) is called real arithmetic(or integer arithmetic). If a combination of arithmetic and real is called mixed mode arithmetic.

2. RELATIONAL OPERATORS : We often compare two quantities and depending on their relation take certain decisions for that comparison we use relational operators.

operator	meaning
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
==	is equal to
!=	is not equal to

It is the form of
ae-1 relational operator ae-2

3. LOGICAL OPERATORS : An expression of this kind which combines two or more relational expressions is termed as a logical expressions or a compound relational expression. The operators and truth values are

op-1	op-2	op-1 && op-2	op-1 || op-2
non-zero	non-zero	1	1
non-zero	0	0	1
0	non-zero	0	1
0	0	0	0

op-1	!op-1
non-zero	zero
zero	non-zero

5. ASSIGNMENT OPERATORS : They are used to assign the result of an expression to a variable. The assignment operator is '='.

v op=exp

v is variable

op binary operator

exp expression

op= short hand assignment operator

short hand assignment operators

use of simple assignment operators	use of short hand assignment operators
a=a+1	a+=1
a=a-1	a-=1
a=a%b	a%=b

6. INCREMENT AND DECREMENT OPERATORS :

++ and == are called increment and decrement operators used to add or subtract. Both are unary and as follows

++m or m++

--m or m--

The difference between ++m and m++ is

if m=5; y=++m then it is equal to m=5;m++;y=m;

if m=5; y=m++ then it is equal to m=5;y=m;m++;

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

exp1 ? exp2 : exp3;

It work as

if exp1 is true then exp2 else exp3

8. BIT WISE OPERATORS : C supports special operators known as bit wise operators for manipulation of data at bit level. They are not applied to float or double.

operator	meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
~	one's complement

9. SPECIAL OPERATORS : These operators which do not fit in any of the above classification are ,(comma), sizeof, Pointer operators(& and *) and member selection operators (. and ->). The comma operator is used to link related expressions together.

sizeof operator is used to know the sizeof operand.

```
/* programs to exhibit the use of operators   */

#include<stdio.h>
main()
{
   int sum, mul, modu;
   float sub, divi;
   int i,j;
   float l, m;
  printf("Enter two integers ");
  scanf("%d%d",&i,&j);
  printf("Enter two real numbers");
  scanf("%f%f",&l,&m);
  sum=i+j;
  mul=i*j;
  modu=i%j;
  sub=l-m;
  divi=l/m;
  printf("sum is %d", sum);
  printf("mul is %d", mul);
  printf("Remainder is %d", modu);
  printf("subtraction of float is %f", sub);
  printf("division of float is %f", divi);
}


/* program to implement relational and logical   */
```

```
#include<stdio.h>
main()
{
        int i, j, k;
        printf("Enter any three numbers ");
        scanf("%d%d%d", &i, &j, &k);
        if((i<j)&&(j<k))
                printf("k is largest");
        else if i<j || j>k
                {
                        if i<j && j >k
                                printf("j is largest");
                        else
                                printf("j is not largest of all");
                }
}

/* program to implement increment and decrement operators  */

#include<stdio.h>
main()
{
        int i;
        printf("Enter a number");
        scanf("%d", &i);
        i++;
        printf("after incrementing  %d ", i);
        i--;
        printf("after decrement %d", i);
}


/* program using ternary operator and assignment  */
#include<stdio.h>
main()
{
        int i,j,large;
        printf("Enter two numbers ");
        scanf("%d%d",&i,&j);
        large=(i>j)?i:j;
        printf("largest of two is  %d",large);
}
```

EXPRESSIONS :

An expression is a sequence of operands and operators that reduces to a single value. Expression 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.E.g: 2 + 3 is a simple expression whose value is 5.A complex expression contains more that one operator. E.g: 2 + 3 * 2. To evaluate a complex expression we reduce it to a series of simple expressions. In this first we will evaluate the simple expression 3 * 2 (6)and then the expression 2 + 6,giving a result of 8.

The order in which the operators in a complex expression are evaluated is determined by a set of priorities known as **precedence,** the higher the precedence ,the earlier the expression containing the operator is evaluated. If two operators with the same precedence occur in a complex expression ,another attribute of an operator ,its associativety ,takes control.**Associativity** is the parsing direction used to evaluate the expression. It can be either left-to-right or right-to-left .When two operators with the same precedence occur in an expression and their associativity is left-to-right ,the left operator is evaluated first. For example ,in the expression 3*4/6 ,there are two operators multiplication and division ,with the same precedence and left-to-right associativity .Therefore the multiplication is evaluated before the division .

The following table shows the precedence and associativity of operators:

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

ARITHMETIC EXPRESSIONS :

It is a combination of variables, constants and operators arranged according to the syntax of C language.

Some examples

A * B – C

(M + N) * (X + Y)

Evaluation of expressions : Expressions are evaluated using an assignment statement of the form

Variable = expression

Eg x = x*y + z-3 *(z *y)

Precedence of arithmetic expressions is used to evaluate a expression to provide un ambiguous way of solving an expression. The highest precedence operator is evaluated then next highest precedence operator until no operator is present.

The precedence or priorities are as follows

High   * / %

Low + -

An **expression is evaluated** in left to right and value is assigned to variable in left side of assignment operator.

```
/* program to demonstrate evaluation of expressions */
#include<stdio.h>
main()
{
        float a,b,c,x,y,z;
        a=9;b=23;c=3;
        x=a-b/3+c*2-1;
        y=a-b/(3+c)*(2-1);
        z=a-(b/(3+c)*2)-1;
        printf("values of x,y,z are %d%d%d",x,y,z);
}
```

TYPE CONVERSION:

In an expression that involves two different data types ,such as multiplying an integer and a floating point number to perform these evaluations ,one of the types must be converted.
We have two types of conversions

1.Implicit Type Conversion
2.Explicit Type Conversion

IMPLICIT TYPE CONVERSION :

When the types of the two operands in a binary expression are different automatically converts one type to another .This is known as implicit type conversion .

EXPLICIT TYPE CONVERSION :

Explicit type conversion uses the unary cast operator ,to convert data from one type to another. To cast data from one type to another ,we specify the new type in parentheses before the value we want converted.

For example ,to convert an integer ,a , to a float, we code the expression like

```
(float) a
```

INPUT AND OUTPUT STATEMENTS :

The simplest of input operator is getchar to read a single character from the input device.

```
        varname=getchar();
```

you need to declare varname.

The simplest of output operator is putchar to output a single character on the output device.

```
        putchar(varname)
```

The getchar() is used only for one input and is not formatted. Formatted input refers to an input data that has been arranged in a particular format, for that we have scanf.

```
        scanf("control string", arg1, arg2,...argn);
```

Control string specifies field format in which data is to be entered.
arg1, arg2... argn specifies address of location or variable where data is stored.

```
        eg  scanf("%d%d",&a,&b);
                %d   used for integers
                %f           floats
                %l           long
                %c           character
```

for formatted output you use printf

```
        printf("control string", arg1, arg2,...argn);
```

```
/* program to exhibit i/o  */
#include<stdio.h>
main()
{
        int a,b;
        float c;
        printf("Enter any number");
        a=getchar();
        printf("the char is ");
        putchar(a);
        printf("Exhibiting the use of scanf");
        printf("Enter three numbers");
        scanf("%d%d%f",&a,&b,&c);
        printf("%d%d%f",a,b,c);
}
```

STATEMENTS AND BLOCKS :

A statement causes an action to be performed by the program. It translates directly in to one or more executable computer instructions.

STATEMENT TYPES:

1.NULL STATEMENT :

The null statement is just a semicolon (the terminator).

```
Eg:
                    ;            //null statement
```

Although they do not arise often, there are syntactical situations where we must have a statement but no action is required .In these situations we use the null statement.

2.EXPRESSION STATEMENT :

An expression is turned in to a statement by placing a semicolon(;)after it.

```
expression;                 //expression statement

        Eg:   a=2;
```

3.RETURN STATEMENT :

A return statement terminates a function. All functions ,including main, must have a return statement. Where there is no return statement at the end of the function ,the system inserts one with a void return value.

```
return expression;              //return statement
```

The return statement can return a value to the calling function. In case of main ,it returns a value to the operating system rather than to another function. A return value of zero tells the operating system that the program executed successfully.

4.COMPOUND STATEMENTS:

A compound statement is a unit of code consisting of zero or more statements .It is also known as a **block**. The compound statement allows a group of statements to become one single entity.

A compound statement consists of an opening brace ,an optional declaration and definition section ,and an optional statement section ,followed by a closing brace.

```
Eg:
    {
            //Local Declarations
```

```
        int x;
        int y;
        int z;
        //Statements
        x=1;
        y=2;
        …
}       //End Block
```

IF AND SWITCH STATEMENTS :

We have a number of situations where we may have to change the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met.

The if statement is a two way decision statement and is used in conjunction with an expression. It takes the following form

If(test expression)

If the test expression is true then the statement block after if is executed otherwise it is not executed

```
if (test expression)
{
        statement block;

}
statement–x ;
```

only statement–x is executed.

```
/* program for if  */
#include<stdio.h>
main()
{
        int a,b;
        printf("Enter two numbers");
        scanf("%d%d",&a,&b):
        if a>b
                printf(" a is greater");
        if b>a
                printf("b is greater");
}
```

The if –else statement:

If your have another set of statement to be executed if condition is false then if-else is used

```
if (test expression)
{
        statement block1;
}
else
{
        statement block2;
}
statement –x ;


/* program for if-else  */

#include<stdio.h>
main()
{
        int a,b;
        printf(“Enter two numbers”);
        scanf(“%d%d”,&a,&b):
        if a>b
                printf(“ a is greater”)
        else
                printf(“b is greater”);
}
```

Nesting of if..else statement :

If more than one if else statement

```
if(text cond1)
{
        if (test expression2
        {
                statement block1;
        }
        else
        {
                statement block 2;
        }
}
```

```
else
{
        statement block2;
}
statement-x ;
```

if else ladder

```
        if(condition1)
                statement1;
        else if(condition2)
                statement 2;
        else if(condition3)
                statement n;
        else
                default statement.
        statement-x;
```

The nesting of if-else depends upon the conditions with which we have to deal.

THE SWITCH STATEMENT :

If for suppose we have more than one valid choices to choose from then we can use switch statement in place of if statements.

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

In case of

```
        if(cond1)
```

```
                {
                statement-1
                }
        if (cond2)
                {
                        statement 2
                }
```

```
/*   program to implement switch   */
#include<stdio.h>
main()
{
         int marks,index;
        char grade[10];
        printf("Enter your marks");
        scanf("%d",&marks);
        index=marks/10;
        switch(index)
        {
                case 10 :
                case 9:
                case 8:
                case 7:
                case 6: grade="first";
                        break;
                case 5 : grade="second";
                        break;
                case 4 : grade="third";
                        break;
                default : grade ="fail";
                        break;
        }
        printf("%s",grade);
}
```

LOOPING :

Some times we require a set of statements to be executed repeatedly until a condition is met.

We have two types of looping structures. One in which condition is tested before entering the statement block called entry control.

The other in which condition is checked at exit called exit controlled loop.

WHILE STATEMENT :

```
While(test condition)
{
        body of the loop
}
```

It is an entry controlled loop. The condition is evaluated and if it is true then body of loop is executed. After execution of body the condition is once again evaluated and if is true body is executed once again. This goes on until test condition becomes false.

```
/* program for while   */
#include<stdio.h>
main()
{
        int count,n;
        float x,y;
        printf("Enter the values of x and n");
        scanf("%f%d",&x,&n);
        y=1.0;
        count=1;
        while(count<=n)
        {
                y=y*x;
                count++;
        }
        printf("x=%f; n=%d; x to power n = %f",x,n,y);
}
```

DO WHILE STATEMENT :

The while loop does not allow body to be executed if test condition is false. The do while is an exit controlled loop and its body is executed at least once.

```
do
{
        body
}while(test condition)
```

```
/* printing multiplication table  */

#include<stdio.h>
#define COL 10
#define ROW 12
main()
{
        int row,col,y;
        row=1;
        do
        {
                col=1;
                do
                {
                        y=row*col;
                        printf("%d",y);
                        col=col+1;
                }while(col<=COL);
                printf("\n");
                row=row+1;
        }while(row<=ROW);
}
```

THE FOR LOOP :

It is also an entry control loop that provides a more concise structure

```
        for(initialization; test control; increment)
        {
                        body of loop
        }
```

```
/*  program of for loop */
#include<stdio.h>
main()
{
        long int p;
        int n;
        double q;
        printf("2 to power n   ");
        p=1;
        for(n=0;n<21;++n)
        {
                if(n==0)
                        p=1;
                else
                        p=p*2;
                q=1.0/(double)p;
                printf("%101d%10d",p,n);
        }
}
```

BREAK STATEMENT:

This is a simple statement. It only makes sense if it occurs in the body of a switch, do, while or for statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the break. Its use is widespread in switch statements, where it is more or less essential to get the control .

The use of the break within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example.

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

    for(i = 0; i < 10000; i++){
        if(getchar() == 's')
            break;
        printf("%d\n", i);
    }
    exit(EXIT_SUCCESS);
}
```

It reads a single character from the program's input before printing the next in a sequence of numbers. If an ‘s’ is typed, the break causes an exit from the loop.

If you want to exit from more than one level of loop, the break is the wrong thing to use.

CONTINUE STATEMENT:

This statement has only a limited number of uses. The rules for its use are the same as for break, with the exception that it doesn't apply to switch statements. Executing a continue starts the next iteration of the smallest enclosing do, while or for statement immediately. The use of continue is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behaviour) doesn't happen.

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

    for(i = -10; i < 10; i++){
        if(i == 0)
            continue;
        printf("%f\n", 15.0/i);
        /*
         * Lots of other statements .....
         */
    }
    exit(EXIT_SUCCESS);
}
```

The continue can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. continue has no special meaning to a switch statement, where break does have. Inside a switch, continue is only valid if there is a loop that encloses the switch, in which case the next iteration of the loop will be started.

There is an important difference between loops written with while and for. In a while, a continue will go immediately to the test of the controlling expression. The same thing in a for will do two things: first the update expression is evaluated, then the controlling expression is evaluated.

GOTO AND LABELS:

In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a *label* when you use a goto; this example shows both.

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own ‘name space’ so they can't clash with the names of variables or functions. The name space only exists for the function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a goto statement.

Labels must be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */
}
```

The goto works in an obvious way, jumping to the labelled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

It's hard to give rigid rules about the use of gotos but, as with the do, continue and the break (except in switch statements), over-use should be avoided. More than one goto every 3–5 functions is a symptom that should be viewed with deep suspicion.

IMPORTANT QUESTIONS UNIT - II

1. Write the various steps involved in executing a C program and illustrate with the help of flow chart?

2. What is the difference between break and continue statements ? Explain with examples.

3. What is the purpose of goto statement? How is the associated target statement identified?

4.(a) What are constants?

(b) Name the different data types that C supports and explain them in detail.

5. (a) What is meant by looping? Describe any two different forms of looping with Examples.

(b) Write a program to print the following outputs using for loop.

```
i)      1
        2  2
        3  3  3
        4  4  4  4
```

```
ii)           1
          2       2
      3       3       3
  4       4       4       4
```

6. What are the logical operators used in C and illustrate with examples?

7. Whar is the purpose of switch statement ? How does this statement differ from the other statements?

8. (a) What is an Expression ? What kind of information is represented by an Expression?

(b) What is an Operator? Describe several different types of operators that are included within the C language with an example each?

9. What are the different types of control statements available in C. Explain them with an example.

10. (a) What is the difference between signed integer and unsigned integer in terms of memory and range?

(b) Explain the basic structure of C program?

11. Explain the following and illustrate it with an example?

(a) Increment and Decrement Operator

(b) Conditional Operator

(c) Bitwise Operator

(d) Assignment operator

12. State the rules that are applied while evaluating expression in automatic type conversion?

13. (a) What is a String constant? How do string constants differ from character constants? Do string constants represent numerical values?

(b) Summarize the standard escape sequences in C? Describe them.

(c) What is a variable. How can variables be characterized? Give the rules for variable declaration.

(d) What is a purpose of type declarations? What are the components of type declaration?

14. (a) Write a program to determine and print the sum of the following harmonic series for a given value of n:

1+1/2+1/3+1/4+…..+1/n.

(b) An electric power distribution company charges its domestic consumers as follows:

Consumption Units	Rate of Charge
0-200	Rs. 0.50 per unit
201-400	Rs.100 plus Rs.0.65 per unit excess of 200
401-600	Rs.230 plus Rs.0.80 per unit excess of 400.

Write a C program that reads the customer number and power consumed and prints the amount to be paid by the customer.

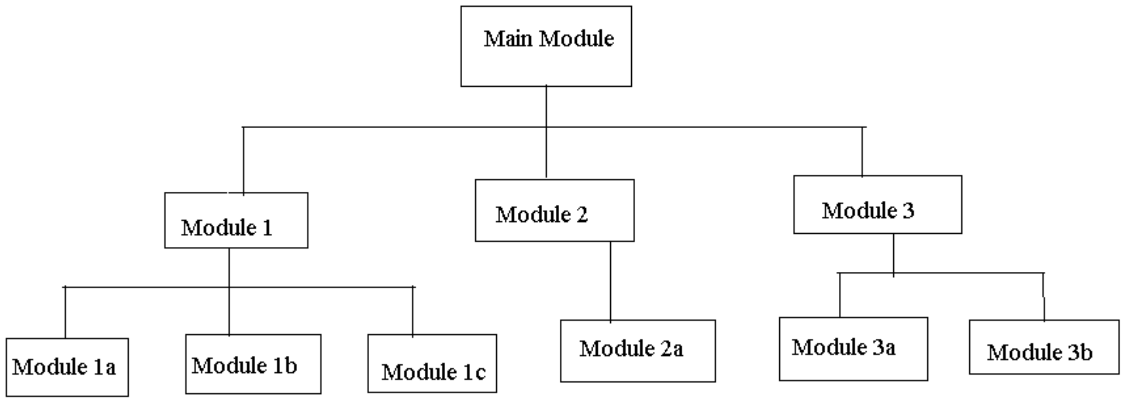
UNIT – III

DESIGNING STRUCTURED PROGRAMS:

The planning for large programs consists of first understanding the problem as a whole, second breaking it into simpler, 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**.

The principles of top-down design and structured programming dictate that a program should be divided into a main module and its related modules. Each module is in turn divided into sub-modules until the resulting modules are intrinsic; that is, until they are implicitly understood without further division.

Top-down design is usually done using a visual representation of the modules known as a structure chart. The structure chart shows the relation between each module and its sub-modules. The structure chart is read top-down, left-right. First we read Main Module. Main Module represents our entire set of code to solve the problem.



Structure Chart

Moving down and left, we then read Module 1. On the same level with Module 1 are Module 2 and Module 3. The Main Module consists of three sub-modules. At this level, however we are dealing only with Module 1. Module 1 is further subdivided into three modules, Module 1a, Module 1b, and Module 1c. To write the code for Module 1, we need to write code for its three sub-modules.

The Main Module is known as a calling module because it has sub-modules. Each of the sub-modules is known as a called module. But because Modules 1, 2, and 3 also have sub-modules, they are also calling modules; they are both called and calling modules.

Communication between modules in a structure chart is allowed only through a calling module. If Module 1 needs to send data to Module 2, the data must be passed through the calling module, Main Module. No communication can take place directly between modules that do not have a calling-called relationship.

How can Module 1a send data to Module 3b?

It first sends data to Module 1, which in turn sends it to the Main Module, which passes it to Module 3, and then on to Module 3b.

The technique used to pass data to a function is known as parameter passing. The parameters are contained in a list that is a definition of the data passed to the function by the caller. The list serves as a formal declaration of the data types and names.

FUNCTIONS :

A function is a self contained program segment that carries out some specific well defined tasks.

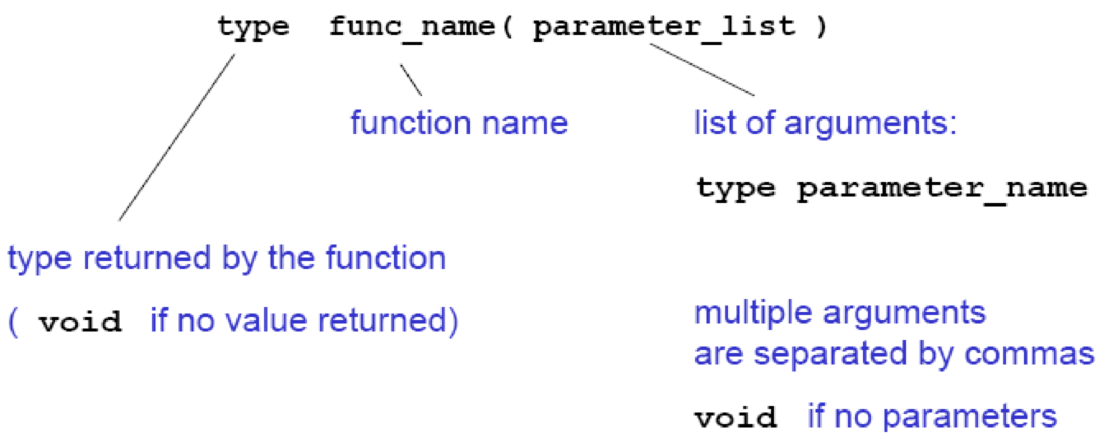
Advantages of functions:

1. Write your code as collections of small functions to make your program modular
2. Structured programming
3. Code easier to debug
4. Easier modification
5. Reusable in other programs

Function Definition :

```
type func_name( parameter list )
{
declarations;
statements;
}
```

FUNCTION HEADER :


type func_name(parameter_list)
function name
list of arguments:
type parameter_name
type returned by the function
(void if no value returned)
multiple arguments are separated by commas
void if no parameters

Examples:

```
int fact(int n)

void error_message(int errorcode)

double initial_value(void)

int main(void)
```

Usage:

```
a = fact(13);

error_message(2);

x=initial_value();
```

FUNCTION PROTOTYPES

If a function is not defined before it is used, it must be declared by specifying the return type and the types of the parameters.

double sqrt(double);

Tells the compiler that the function **sqrt()** takes an argument of type double and returns a double. This means, incidentally, that variables will be cast to the correct type; so **sqrt(4)** will return the correct value even though **4** is **int** not **double**. These function prototypes are placed at the top of the program, or in a separate header file, **file.h**, included as

#include "file.h"

Variable names in the argument list of a function declaration are optional:

void f (char, int);

void f (char c, int i); /*equivalent but makes code more readable */

If all functions are defined before they are used, no prototypes are needed. In this case, **main()** is the last function of the program.

SCOPE RULES FOR FUNCTIONS :

Variables defined within a function (including main) are local to this function and no other function has direct access to them. The only way to pass variables to function is as parameters. The only way to pass (a single) variable back to the calling function is via the return statement

Ex:

```
int func (int n)
{
printf("%d\n",b);
return n;
}//b not defined locally!

int main (void)
{
int a = 2, b = 1, c;
c = func(a);
return 0;
}
```

FUNCTION CALLS :

When a function is called, expressions in the parameter list are evaluated (in no particular order!) and results are transformed to the required type. Parameters are copied to local variables for the function and function body is executed when return is encountered, the function is terminated and the result (specified in the return statement) is passed to the calling function (for example main).

Ex:

```
int fact (int n)
{
int i, product = 1;
for (i = 2; i <= n; ++i)
product *= i;
return product;
}

int main (void)
{
```

```
int i = 12;
printf("%d",fact(i));
return 0;
}
```

PARAMETER PASSING :

Parameter passing mechanism in 'C' is of two types.

1. Call by Value
2. Call by Reference.

The process of passing the actual value of variables is known as Call by Value.

The process of calling a function using pointers to pass the addresses of variables is known as Call by Reference. The function which is called by reference can change the value of the variable used in the call.

Example of Call by Value:

```
#include <stdio.h>
void swap(int,int);
main()
{
        int a,b;
        printf("Enter the Values of a and b:");
        scanf("%d%d",&a,&b);
        printf("Before Swapping \n");
        printf("a = %d \t b = %d", a,b);
        swap(a,b);
        printf("After Swapping \n");
        printf("a = %d \t b = %d", a,b);
}


void swap(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

Example of Call by Reference:

```
#include<stdio.h>
main()
{
        int a,b;
        a = 10;
        b = 20;
        swap (&a, &b);
        printf("After Swapping \n");
        printf("a = %d \t b = %d", a,b);
}
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

STORAGE CLASSES :

In 'C' a variable can have any one of four Storage Classes.

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

SCOPE :

The Scope of variable determines over what parts of the program a variable is actually available for use.

LONGEVITY :

Longevity refers to period during which a variable retains a given value during execution of a program (alive). So Longevity has a direct effect on utility of a given variable.

The variables may also be broadly categorized depending on place of their declaration as internal(local) or external(global). Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.

AUTOMATIC VARIABLES :

They are declared inside a function in which they are to be utilized. They are created when function is called and destroyed automatically when the function is exited, hence the name automatic. Automatic Variables are therefore private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables.

By default declaration is automatic. One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program.

```
#include<stdio.h>
main()
{
        int m=1000;
        func2();
        printf("%d\n",m);
}


func1()
{
        int m=10;
        printf("%d\n",m);
}
func2()
{
        int m=100;
        func1();
        printf("%d",m);
}
```

First, any variable local to main will normally live throughout the whole program, although it is active only in main.
Secondly, during recursion, nested variables are unique auto variables, a situation similar to function nested auto variables with identical names.

EXTERNAL VARIABLES :

Variables that are both alive and active throughout entire program are known as external variables. They are also known as Global Variables. In case a local and global have same name local variable will have precedence over global one in function where it is declared.

```
#include<stdio.h>
int x;
main()
{
        x=10;
        printf("%d",x);
        printf("x=%d",fun1());
        printf("x=%d",fun2());
        printf("x=%d",fun3());


}

fun1()
{
        x=x+10;
        return(x);
}
fun2()
{
        int x;
        x=1;
        return(x);
}
fun3()
{
x=x+10;
return(x);
}
```

An extern within a function provides the type information to just that one function.

STATIC VARIABLES :

The value of Static Variable persists until the end of program. A variable can be declared Static using Keyword Static like Internal & External Static Variables are differentiated depending whether they are declared inside or outside of auto variables, except that they remain alive throughout the remainder of program.

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

```
{
        int I;
        for (I=1;I<=3;I++)
        stat();
}
stat()
{
        static int x=0;
        x=x+1;
        printf("x=%d\n",x);
}
```

REGISTER VARIABLES :

We can tell the Compiler that a variable should be kept in one of the machines registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping frequently accessed variables in register will lead to faster execution

Syntax:

```
register  int Count.
```

USER DEFINED FUNCTIONS:

Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result task of debugging, testing and maintaining becomes difficult. If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit, these subprograms called "functions" are much easier to understand debug and test.

There are times when some types of operation for calculation is repeated many times at many points through out a program. For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can be called and used whenever required.

The advantages of using functions are

1. It facilitates top-down modular programming.
2. The length of a source program can be reduced by using functions at appropriate places.
3. It is easy to locate and isolate a faculty function for further investigations.

4. A function may be used by many other programs.

A function is a self-contained block of code that performs a particular task. Once a function has been designed and packed it can be treated as a "black box" that takes some data from main program and returns a value. The inner details of the program are invisible to the rest of program.

The form of the C functions.

```
function-name ( argument list )
argument declaration;
{
     local variable declarations;
    executable statement 1;
    executabie      statement 2;
    - - - - - - - - - -
    - - - - - - - - - -
    - - - - - - - - - -
    return (expression) ;
 }
```

The return statement is the mechanism for returning a value to the calling function. All functions by default returns int type data. we can force a function to return a particular type of data by using a type specifier in the header.

A function can be called by simply using the function name in the statement.

A function depending on whether arguments are present or not and whether a value is returned or not may belong to.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

1. Functions with no arguments and no return values :

When a function has no arguments, it does not receive any data from calling function. In effect, there is no data transfer between calling function and called function.

```
#include<stdio.h>
main()
{
    printline();
    value();
    printline();
}
printline()
```

```
{
        int i;
        for(i=1;i<=35;i++0)
                printf(“%c”,”-“);
                printf(“\n”);
}
value()
{
        int year, period;
        float inrate,sum,principal;
        printf(“Enter Principal, Rate,Period”);
        scanf(“%f%f%f”,&principal,&inrate,&period);
        sum=principal;
        year=1;
        while(year<=period)
        {
                sum=sum*(1+inrate);
                year=year+1;
        }
        printf(“%f%f%d%f”,principal,inrate,period,sum);
}
```

2. Arguments but no return values :

The function takes argument but does not return value.
The actual (sent through main) and formal(declared in header section) should match in number, type and order.

In case actual arguments are more than formal arguments, extra actual arguments are discarded. On other hand unmatched formal arguments are initialized to some garbage values.

```
#include<stdio.h>
main()
{
        float prin,inrate;
        int period;
        printf(“Enter principal amount, interest”);
        printf(“rate and period\n”);
        scanf(“%f%f%d”,&principal,&inrate,&period);
        printline(‘z’);
        value(principal, inrate, peiod);
        printline(‘c’);
}
```

```
printline(ch)
char ch;
{
        int i;
        for(i=1;i<=52;i++)
                printf("%c",ch);
                printf("\n");
}
```

STANDARD LIBRARY FUNCTIONS AND HEADER FILES:

C functions can be classified into two categories, namely, library functions and user-defined functions .Main is the example of user-defined functions.Printf and scanf belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us where as a user-defined function has to be developed by the user at the time of writing a program.However, a user-defined function can later become a part of the c program library.

ANSI C Standard Library Functions :

The C language is accompanied by a number of library functions that perform various tasks.The ANSI committee has standardized header files which contain these functions.

Some of the Header files are:

<ctype.h>	character testing and conversion functions.
<math.h>	Mathematical functions
<stdio.h>	standard I/O library functions
<stdlib.h>	Utility functions such as string conversion routines memory allocation routines , random number generator,etc.
<string.h>	string Manipulation functions
<time.h>	Time Manipulation functions

MATH.H

The math library contains functions for performing common mathematical operations. Some of the functions are :

abs :	returns the absolute value of an integer x
cos :	returns the cosine of x, where x is in radians
exp:	returns "e" raised to a given power
fabs:	returns the absolute value of a float x

log: returns the logarithm to base e
log10: returns the logarithm to base 10
pow : returns a given number raised to another number
sin : returns the sine of x, where x is in radians
sqrt : returns the square root of x
tan : returns the tangent of x, where x is in radians
ceil : The ceiling function rounds a number with a decimal part up to the next highest integer (written mathematically as $\lceil x \rceil$)
floor : The floor function rounds a number with a decimal part down to the next lowest integer (written mathematically as $\lfloor x \rfloor$)

STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat : Concatenates (i.e., adds) two strings
strcmp: Compares two strings
strcpy: Copies a string
strlen: Calculates the length of a string (not including the null)
strstr: Finds a string within another string
strtok: Divides a string into tokens (i.e., parts)

STDIO.H

Printf: Formatted printing to stdout
Scanf: Formatted input from stdin
Fprintf: Formatted printing to a file
Fscanf: Formatted input from a file
Getc: Get a character from a stream (e.g, stdin or a file)
putc: Write a character to a stream (e.g, stdout or a file)
fgets: Get a string from a stream
fputs: Write a string to a stream
fopen: Open a file
fclose: Close a file

STDLIB.H

Atof: Convert a string to a double (not a float)
Atoi: Convert a string to an int
Exit: Terminate a program, return an integer value
free: Release allocated memory
malloc: Allocate memory

rand: Generate a pseudo-random number
system: Execute an external command

TIME.H

This library contains several functions for getting the current date and time.

Time: Get the system time
Ctime: Convert the result from time() to something meaningful

The following table contains some more header files:

<**assert.h**>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.
<**complex.h**>	A set of functions for manipulating complex numbers. (New with **C99**)
<**ctype.h**>	Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<**errno.h**>	For testing error codes reported by library functions.
<**fenv.h**>	For controlling floating-point environment. (New with **C99**)
<**float.h**>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<**inttypes.h**>	For precise conversion between integer types. (New with **C99**)
<**iso646.h**>	For programming in ISO 646 variant character sets. (New with **NA1**)

<**limits.h**>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<**locale.h**>	For setlocale() and related constants. This is used to choose an appropriate locale.
<**math.h**>	For computing common mathematical functions
<**setjmp.h**>	Declares the macros setjmp and longjmp, which are used for non-local exits
<**signal.h**>	For controlling various exceptional conditions
<**stdarg.h**>	For accessing a varying number of arguments passed to functions.
<**stdbool.h**>	For a boolean data type. (New with **C99**)
<**stdint.h**>	For defining various integer types. (New with **C99**)
<**stddef.h**>	For defining several useful types and macros.
<**stdio.h**>	Provides the core input and output capabilities of the C language. This file includes the venerable `printf` function.
<**stdlib.h**>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.

<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions. (New with **C99**)
<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wctype.h>	For classifying wide characters. (New with **NA1**)

RECURSIVE FUNCTIONS :

Recursion is a repetitive process in which a function calls itself (or) A function is called recursive if it calls itself either directly or indirectly. In C, all functions can be used recursively.

Example: Fibonacci Numbers

A recursive function for Fibonacci numbers (0,1,1,2,3,5,8,13...)

```
/* Function with recursion*/
int fibonacci(int n)
{
if (n <= 1)
return n;
else
return (fibonacci(n-1) + fibonacci(n-2));
}
```

With recursion 1.4 x 10 ^{9} function calls needed to find the 43rd Fibonacci number(which has the value 433494437) .If possible, it is better to write iterative functions.

```
int factorial (int n)  /* iterative version */
{
for ( ; n > 1; --n)
product *= n;
return product;
}
```

C PREPROCESSOR :

C Preprocessor is a program that processes source program before it is passed to the compiler.Preprocessor commands often known as directives.We can write C programs without knowing anything about the preprocessor or its facilities.But preprocessor is such a great convenience that virtually all C programmers rely on it.

FEATURES OF C PREPROCESSOR :

The C program is often known as 'source code'.The preprocessor works on the source code and creates 'Expanded source code'. If the sorce code is stored in a file pr1.C ,then the expanded source code gets stored in a file pr1.I .It is this expanded source code that is sent to the compiler for compilation.

The preprocessor offers several features called preprocessor directives .Each of these preprocessor directives begins with a # symbol.The directives can be placed anywhere in a program but are most ofhen placed at the beginning of a program .

PREPROCESSOR DIRECTIVES:

Preprocessor directives are:

1.Macro expansion
2.File inclusion
3.Conditional compilation
4.Miscellaneous directives

MACRO EXPANSION:

EXAMPLE:

```
#include<stdio.h>
#define UPPER 25
void main()
{
```

```
int I;
for(i=1;i<=UPPER;i++)
printf("\n%d",i);
}
```

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

```
#define UPPER 25
```

This statement is called Macro definition or more commonly just a Macro.
During preprocessing , the preprocessor replaces every occurrence of UPPER in the program with 25.

MACROS WITH ARGUMENTS:

The Macros that we have used so far are called simple macros. Macros can have arguments,just as functions can.

Example:

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

OUTPUT:

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

In this program , whenever the preprocessor finds the phrase AREA(x) it expands it into the statement (3.14 * x * x).The x in the macro template AREA(x) is an argument that matches the x in the macro expansion (3.14 * x * x). The statement AREA(r1) in the program causes the variable r1 to be substituted for x. Thus the statement AREA(r1) is equalent to:

```
(3.14*r1*r1)
```

After the above source code has passed through the preprocessor , the compiler gets to work on will be this:

```
#include<stdio.h>
void main()
{
float r1=6.25,r2=2.5,a;
a=3.14*r1*r1;
printf("Area of circle=%f\n",a);
a=3.14*r2*r2;
printf("Area of circle=%f",a);
}
```

SOME IMPORTANT POINTS TO REMEMBER WHILE WRITING MACROS WITH ARGUMENTS:

(A) Be careful not to leave a blank between the macro template anf its arfument nwhile defining the macro. For example there should be no blank between AREA and (x) in the definition,

```
#define AREA(x) (3.14*x*x)
```

If we write AREA (x) instead of AREA(x), the (x) would become a part of macro expansion,which we certainly don't want .What would happen is ,the template would be expanded to

```
(r1) (3.14*r1*r1)
```

which won't run.

b)The entire macro expansion should be enclosed within parentheses.

Ex:

```
#include<stdio.h>
#define SQUARE(n) n*n
void main()
{
        int j;
        j=64/SQUARE(4);
        printf("j=%d", j);
}
```

The output of the above program will be:

```
J=64
```

Where as ,we expected was j=4;
In this case macro was expanded into
J=64/4*4;

Which yields 64.

c)Macros can be split into multiple lines, with a '\'(back slash) present at the end of each line.

Ex:

```
            #include<stdio.h>
            #define HLINE for(i=0;i<79;i++)\printf("%c",196);
            #define VLINE { \ gotoxy(X,Y);\ printf("%c",179);\ }
            void main()
            {
                    int  i,y;
                    clrscr();
                    gotoxy(1,12);/*position cursor in row x and column y*/
                    HLINE
                    For(y=1;y<25;y++)
                    VLINE(39,y);
}
```

This program draws a vertical and horizontal line in the center of the screen.

d) If for any reason ,you are unable to debug a macro,then you should view the expanded code of the program to see how the macros are getting expanded. If your source code is present in the file pr1.c then the expanded source code would be stored in pr1.I.You need to generate this file at the command prompt by saying:

```
        cpp pr1.c
```

Here CPP stands for C PreProcessor.It generates the expanded source code and stores it in a file called pr1.I.You can now open this file and see the expanded source code.

MACROS VERSUS FUNCTIUONS :

Macro calls are like function calls ,they are not really the same things.The difference between these two are:

1.In a macro call call ,the preprocessor replaces the macro template with its macro expansion ,In function call the contol is passed to a function along with certain arguments.,some calculations are performed in the function and a useful value is returned back from the function.

2. Macros make the program run faster but increase the size of the program,where as functions make the program smaller and compact.

3.If we use a macro hundred times in a program ,the macro expansion goes into sorce code at hundred different places,thus increasing the program size.On the other hand

,if a function is used ,then even if it is called from hundred different places in the program,it would take the same amount of space in the program.

4.Passing parameters to a function and getting back the returned value does take time and would therefore slowdown the program.

5. The macro is simple and it avoids the overhead associated with function calls if we have a fairly large macro and it is used fairly often ,we ought to replace it with a function.

6.If memory space in the machine in which the program is being executed is less,then it may make sense to use functions instead of macros.By doing so,the program may run slower,but will need less memory space.

FILE INCLUSION:

The first and most common job of a preprocessor is File inclusion, the copying of one or more files in toi programs the files are usually header files that contain function and data declaratuions for the program,but they can contain any valid ‘C’ statement.

The preprocessor command is #include,and it has two different formats.
The first format is used to direct the preprocessor to include header files from the system library.In thus format, the name of the header file is enclosed in pointed brackets .
The second format makes the preprocessor look for the header files in the userdefined directory. In this format, the name of the file pathname is enclosed in double quotes.

Example:

```
#include<filename.h>
#include”filepath.h”
```

CONDITIONAL COMPILATION:

Conditional compilation allows us to control the compilation process by including or excluding statements.

TWO WAY COMMANDS:

The two way command tells the preprocessor to select between two choices this is similar to selection.
Syntax:

```
#if expression
//code to be included if expression is true
#else
//code to be included if expression is false
#endif
```

The expression is a constant value that evaluates to zero or non zero.If the value is nonzero ,it is interpreted as true and the code after #if is included.If it is zero,it is interpreted as false and the code after #else is executed.
The #if part (or) the #else part can be empty.

Example:

```
#if expression                     →     #if !expression
#else                                    //code to be included if expression is false
//code   to   be   included   if   expression   is   false                    #endif
#endif
```

```
#if expression                     →  #if expression
//code to be included if expression is true   //code to be included if expression is true
#else                                 #endif
#endif
```

Two abbreviations:

The C language provides two additional macro commands they are

```
#if defined name    ——————————→  # ifdef name

#if !defined name   ——————————→  #ifndef name
```

MULTI-WAY COMMANDS:

Conditional commands can also be multi-way by selecting one of the choices among several. In the preprocessor, we use the #elif command.

Syntax:

```
        #if expression1
        //code to be included if expression1  is true
        #elif expression2
        //code to be included if expression2 is true
        #else
        //code to be included if both expressions are false
        #endif
```

MISCELLANEOUS DIRECTIVES:

There are two or more preprocessor directives available,though they are not very commonly used . They are

A) #undef
B) #pragma

#undef directive: In order to undefined a macro that has been earlier #defined , we use #undef directive.

Syntax:

```
#undef macro template
```

example:

```
#undef Pentium
```

The above example cause the definition of Pentium to be removed from the system.All subsequent #ifdef Pentium statements would evaluate to false.

#pragma directive: This directive is another special purpose directive that you can use to turn on or off certain features.pragmas vary from one compiler to another.
Example:

```
#pragma tokens
```

This command causes the compiler to perform implementation defined actions.

ARRAYS :

An array is a group of related data items that share a common name.
Ex:- Students
The complete set of students are represented using an array name students. A particular value is indicated by writing a number called index number or subscript in brackets after array name. The complete set of value is referred to as an array, the individual values are called elements.

ONE – DIMENSIONAL ARRAYS :

A list of items can be given one variable index is called single subscripted variable or a one-dimensional array.

The subscript value starts from 0. If we want 5 elements the declaration will be

```
int  number[5];
```

The elements will be number[0], number[1], number[2], number[3], number[4] There will not be number[5]

Declaration of One - Dimensional Arrays :

Type variable – name [sizes];

Type – data type of all elements Ex: int, float etc.,
Variable – name – is an identifier
Size – is the maximum no of elements that can be stored.
Ex:- float avg[50]
This array is of type float. Its name is avg. and it can contains 50 elements only. The range starting from 0 – 49 elements.

Initialization of Arrays :

Initialization of elements of arrays can be done in same way as ordinary variables are done when they are declared.

Type array name[size] = {List of Value};
Ex:- int number[3]={0,0,0};

If the number of values in the list is less than number of elements then only that elements will be initialized. The remaining elements will be set to zero automatically.
Ex:- float total[5]= {0.0,15.75,-10};

The size may be omitted. In such cases, Compiler allocates enough space for all initialized elements.

```
int counter[ ]= {1,1,1,1};
```

```
/* Program Showing one dimensional array */

#include<stdio.h>
main()
{
int i;
float x[10],value,total;
printf("Enter 10 real numbers\n");
        for(i=0;i<10;i++)
        {
                scanf("%f",&value);
                x[i]=value;
        }
        total=0;
        for(i=0;i<10;i++)
                total=total+x[i]
        for(i=0;i<10;i++)
                printf("x[%2d]=%5.2f\n",I+1,x[I]);
                printf("total=%0.2f",total);
}
```

TWO – DIMENSIONAL ARRAYS:

To store tables we need two dimensional arrays. Each table consists of rows and columns. Two dimensional arrays are declare as
type array name [row-size][col-size];

```
/* Write a program Showing 2-DIMENSIONAL ARRAY */

/* SHOWING MULTIPLICATION TABLE */

#include<stdio.h>
#include<math.h>
#define  ROWS 5
#define  COLS 5
main()
{
        int row,cols,prod[ROWS][COLS];
        int i,j;

        printf("Multiplication table");
        for(j=1;j< =COLS;j++)
                printf("%d",j);
        for(i=0;i<ROWS;i++)
        {
                row = i+1;
                printf("%2d|",row);
                for(j=1;j < = COLS;j++)
                {
                        COLS=j;
                        prod[i][j]= row * cols;
                        printf("%4d",prod[i][j]);
                }
```

```
          }
}
```

INITIALIZING TWO DIMENSIONAL ARRAYS:

They can be initialized by following their declaration with a list of initial values enclosed in braces.

Ex:- int table[2][3] = {0,0,0,1,1,1};

Initializes the elements of first row to zero and second row to one. The initialization is done by row by row. The above statement can be written as

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

When all elements are to be initialized to zero, following short-cut method may be used.

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

MULTI-DIMENSIONAL ARRAYS:

C allows arrays of three or more dimensions. The exact limit is determined by Compiler.

type array-names[s1][s2][s3] - - - - - [s_n];

where s_i is size of dimension.

Ex:- int Survey[3][5][2];

```
#include <stdio.h>

void printArray( const int a[][ 3 ] );

int main()
{
  int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
  int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
  int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

  printf( "Values in array1 by row are:\n" );
  printArray( array1 );

  printf( "Values in array2 by row are:\n" );
  printArray( array2 );
  printf( "Values in array3 by row are:\n" );
  printArray( array3 );

  return 0;

}

void printArray( const int a[][ 3 ] )
{
  int i;
  int j;

  for ( i = 0; i <= 1; i++ ) {
    for ( j = 0; j <= 2; j++ ) {
      printf( "%d ", a[ i ][ j ] );
    }

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

APPLICATIONS OF ARRAYS :

1.Frequency arrays with their graphical representations.
2.Random number permutations.

FREQUENCY ARRAYS :

Two common statistical applications that use arrays are frequency distributions and histograms. A frequency array shows the number of elements with an identical value found in a series of numbers.

For example ,suppose we have taken a sample of 100 values between 0 and 19.We want to know how many of the values are 0,how many are 1,how many are 2,and so forth up through 19.

We can read these numbers into an array called numbers. Then we create an array of 20 elements that will show the frequency of each number in the series.

One way to do it is to assign the value from the data array to an index and then use the index to access the frequency array.

```
F=numbers[i];
 Frequency[F]++;
```

Since an index is an expression ,however ,we can simply use the value from our data array to index us in to the frequency array .The value of numbers[i] is determined first ,and then that value is used to index in to frequency.

```
Frequency[numbers[i]]++;
```

Fig-8-14

HISTOGRAMS:

A histogram is a pictorial representation of a frequency array .Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart. For example ,the following figure is a histogram for a set of numbers in the range 0 …19.In this example, asterisks (*) are used to build the bar. Each asterisk represents one occurrence of the data value.

```
0     0
1     4 * * * *   ————————(four 1s)
2     7 * * * * * * *
3     7 * * * * * * *————————(seven 3s)
.
.
.
18    2 * *
19    0 ————————————————(zero 19s)
```

RANDOM NUMBER PERMUTATIONS :

A random number permutation is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 10 numbers, the values from 0 to 9 would all be included with no duplicates.

Generating random numbers:

To generate a random integral in a range x to y,we must first scale the number and then ,if x is greater than 0 ,shift the number within the range. We scale the number using the modulus operator.

Ex: To produce a random number in the range 0 …50,we simply scale the random number and scaling factor must be one greater than the highest number needed.

```
rand ( ) %51
```

modulus works well when our range starts at 0.for example ,suppose we want a random number between 10 and 20.for this we will use one formula

```
range =  ( 20 - 10 ) +1;
randno = rand ( ) % range + 10;
```

To generate a permutation ,we need to eliminate the duplicates. Using histogram concept we can solve the problem most efficiently by using two arrays. The first array contains the random numbers. The second array contains a logical value that indicates whether or not the number represented by its index has been placed in the random number array.

Only the first five random numbers have been placed in the permutation. For each random number in the random number array ,its corresponding location in the have-random array is set to 1. Those locations representing numbers that have not yet been generated are still set to 0.

IMPORTANT QUESTIONS UNIT - III

1. (a) What is a Preprocessor directive?

   (b) Distinguish between function and Preprocessor directive?

   (c) What is the significance of Conditional Compilation?

(d) How does the undefining of a pre-defined macro done?

2. (a) Write a program to demonstrate passing an array argument to a function? Consider the problem of finding largest of N numbers defined in an array.
   (b) Write a recursive function power(base, exponent) that when invoked returns base exponent?
3. What do you mean by functions? Give the structure of the functions and explain about the arguments and their return values?
4. (a) Distinguish between the following.
      (i) Actual and Formal arguments?
      (ii) Global and Local variables?
      (iii) Automatic and Static variables?
   (b) Explain in detail about pass by value and pass by reference. Explain with a sample program?
5. (a) Distinguish between formal variable and actual variable.
   (b) Distinguish between Local and Global variable.
   (c) Distinguish between Call by value and Call by reference.
6. (a) What is the syntax followed in the definition of a macro?
   (b) What are the advantages and disadvantages of macros?
   (c) What is meant by conditional compilation? What are its advantages?
7. (a) What are the advantages and disadvantages of recursion?
   (b) Write a C program to find the factors of a given integer using a function.
8. (a) Give some important points while using return statement.
   (b) Write short notes on scope of a variable.
9. (a) Write short notes on auto and static storage classes.
   (b) Write short notes on call by reference.
10. (a) Distinguish between user defined and built-in functions.
    (b) What is meant by function prototype? Give an example for function prototype.
11. (a) Distinguish between getchar and scanf functions for reading strings.
    (b) Write a program to count the number of words, lines and characters in a text.

12. (a) What do you mean by functions? Give the structure of the functions and explain about the arguments and their return values.

    (b) Write a C program that uses a function to sort an array of integers.

13. Define an array. What are the different types of arrays? Explain

14. (a) Write a C program to do matrix multiplications.

    (b) Write in detail about one dimensional and multidimensional arrays. Also write about how initial values can be specified for each type of array.

FILL IN BLANKS:-

1. Array is a group of ________________data items.
2. Array is a _______________________ data type.
3. Arrays can be of __________________data types.
4. The elements or the members of the Array are accessed by _______________ names. (same/different).
5. Elements of an array are differentiated by a single subscript value called _______________.
6. The index value of Array Starts from ________________ number.

7..Arrays are of _______ dimensional and ___________dimensional and ___________ dimensional

UNIT – IV

POINTERS :

One of the powerful features of C is ability to access the memory variables by their memory address. This can be done by using Pointers. The real power of C lies in the proper use of Pointers.

A pointer is a variable that can store an address of a variable (i.e., 112300).We say that a pointer points to a variable that is stored at that address. A pointer itself usually occupies 4 bytes of memory (then it can address cells from 0 to 232-1).

Advantages of Pointers :

1. A pointer enables us to access a variable that is defined out side the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.

Definition :

A variable that holds a physical memory address is called a pointer variable or Pointer.

Declaration :

```
        Datatype * Variable-name;

Eg:-    int *ad;                /* pointer to int */
        char *s;           /* pointer to char */
        float *fp;         /* pointer to float */
        char **s;          /* pointer to variable that is a pointer to char */
```

A pointer is a variable that contains an address which is a location of another variable in memory.

Consider the Statement

```
        p=&i;
```

Here '&' is called address of a variable.
'p' contains the address of a variable i.

The operator & returns the memory address of variable on which it is operated, this is called Referencing.

The * operator is called an indirection operator or dereferencing operator which is used to display the contents of the Pointer Variable.

Consider the following Statements :

```
int *p,x;
x =5;
p= &x;
```

Assume that x is stored at the memory address 2000. Then the output for the following printf statements is :

```
                                                    Output
Printf(“The Value of x is %d”,x);                      5
Printf(“The Address of x is %u”,&x);                2000
Printf(“The Address of x is %u”,p);                 2000
Printf(“The Value of x is %d”,*p);                     5
Printf(“The Value of x is %d”,*(&x));                  5
```

POINTER FUNCTION ARGUMENTS

Function arguments in C are strictly pass-by-value. However, we can simulate pass-by-reference by passing a pointer. This is very useful when you need to Support in/out(bi-directional) parameters (e.g. swap, find replace) Return multiple outputs (one return value isn't enough) Pass around large objects (arrays and structures).

```
/*  Example of swapping a function can't change parameters */
void bad_swap(int x, int y)
{
int temp;
temp = x;
x = y;
y = temp;
}
/* Example of swapping - a function can't change parameters, but if a parameter is a
pointer it can change the value it points to */

void good_swap(int *px, int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
  }

#include <stdio.h>
void bad_swap(int x, int y);
void good_swap(int *p1, int *p2);
main() {
int a = 1, b = 999;
printf("a = %d, b = %d\n", a, b);
bad_swap(a, b);
printf("a = %d, b = %d\n", a, b);
good_swap(&a, &b);
printf("a = %d, b = %d\n", a, b);
}
```

POINTERS AND ARRAYS :

When an array is declared, elements of array are stored in contiguous locations. The address of the first element of an array is called its base address.

Consider the array

2000	2002	2004	2006	2008
a[0]	a[1]	a[2]	a[3]	a[4]

The name of the array is called its base address.

i.e., a and k& a[20] are equal.

Now both a and a[0] points to location 2000. If we declare p as an integer pointer, then we can make the pointer P to point to the array a by following assignment

P = a;

We can access every value of array a by moving P from one element to another.

i.e.,	P	points to 0th element
	P+1	points to 1st element
	P+2	points to 2nd element
	P+3	points to 3rd element
	P +4	points to 4th element

Reading and Printing an array using Pointers :

```
main()
{
        int *a,i;
        printf(“Enter five elements:”);
        for (i=0;i<5;i++)
                scanf(“%d”,a+i);
                printf(“The array elements are:”);
                for (i=o;i<5;i++)
                        printf(“%d”, *(a+i));
}
```

In one dimensional array, a[i] element is referred by

(a+i) is the address of ith element.

* (a+i) is the value at the ith element.

In two-dimensional array, a[i][j] element is represented as

((a+i)+j)

POINTERS AND FUNCTIONS :

Parameter passing mechanism in ‘C’ is of two types.

1.Call by Value
2.Call by Reference.

The process of passing the actual value of variables is known as Call by Value.The process of calling a function using pointers to pass the addresses of variables is known as Call by Reference. The function which is called by reference can change the value of the variable used in the call.

Example of Call by Value:

```
#include <stdio.h>
void swap(int,int);
main()
{
        int a,b;
        printf(“Enter the Values of a and b:”);
        scanf(“%d%d”,&a,&b);
        printf(“Before Swapping \n”);
        printf(“a = %d \t b = %d”, a,b);
        swap(a,b);
        printf(“After Swapping \n”);
        printf("a = %d \t b = %d", a,b);
}


void swap(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

Example of Call by Reference:

```
#include<stdio.h>
main()
{
        int a,b;
        a = 10;
        b = 20;
        swap (&a, &b);
        printf("After Swapping \n");
        printf("a = %d \t b = %d", a,b);
}
void swap(int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

ADDRESS ARITHIMETIC :

Incrementing/Decrementing a pointer variable ,adding and subtracting an integer from pointer variable are all legal and known as pointer arithmetic. Pointers are valid operands in arithmetic expressions ,assignment expressions ,and comparison expressions. However not all the operators normally used in these expressions are valid in conjunction with pointer variable.

A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented(+ +) or decremented(--) ,an integer may be added to a pointer
(+ or +=),an integer may be subtracted from a pointer(- or -=),or one pointer may be subtracted from another.

We can add and subtract integers to/from pointers – the result is a pointer to another element of this type

Ex : int *pa; char *s;

```
s-1 →points to char before s (1 subtracted)
pa+1→  points to next int (4 added!)
s+9 →points to 9th char after s (9 added)
++pa→  increments pa to point to next int
```

NULL POINTER :

'Zero' is a special value we can assign to a pointer which does not point to anything most frequently, a symbolic constant NULL is used. It is guaranteed, that no valid address is equal to 0.The bit pattern of the NULL pointer does not have to contain all zeros usually it does or it depends on the processor architecture. On many machines, dereferencing a NULL pointer causes a segmentation violation.

NULL ptr is not the same as an EMPTY string.

```
const char* psz1 = 0;
const char* psz2 = "";
assert(psz1 != psz2);
```

Always check for NULL before dereferencing a pointer.

```
if (psz1)
/* use psz1 */
sizeof(psz1)    // doesn't give you the number of elements in psz1. Need
additional size variable.
```

VOID POINTER :

In C ,an additional type void *(void pointer) is defined as a proper type for generic pointer. Any pointer to an object may be converted to type void * without loss of information. If the result is converted back to the original type ,the original pointer is recovered .

Ex:

```
main()
{
        void *a;
        int n=2,*m;
        double d=2.3,*c;
        a=&n;
        m=a;
        printf("\n%d %d %d",a,*m,m);
        a=&d;
        c=a;
        printf("\n%d %3.1f %d",a,*c,c);
}
```

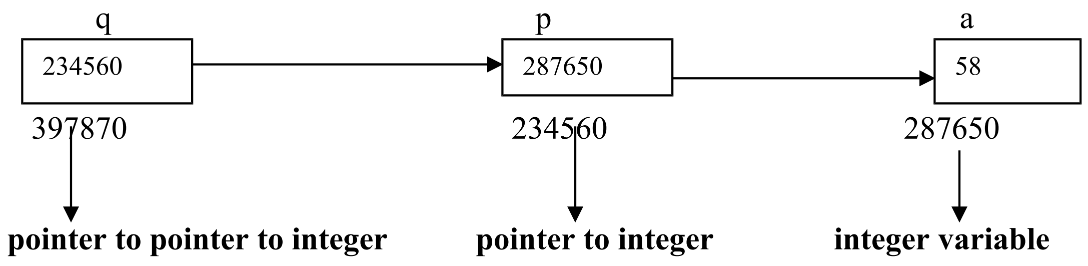
In the above program a is declared as a pointer to void which is used to carry the address of an int(a=&n)and to carry the address of a double(a=&d) and the original pointers are recovered with out any loss of information.

POINTERS TO POINTERS :

So far ,all pointers have been pointing directely to data.It is possible and with advanced data structures often necessary to use pointers to that point to other pointers.
For example,we can have a pointer pointing to a pointer to an integer.This two level indirection is seen as below:

```
//Local declarations
int a;
int* p;
int **q;
```

Ex:



```
//statements
a=58;
p=&a;
q=&p;
printf("%3d",a);
printf("%3d",*p);
```

printf(“%3d”,**q);

There is no limit as to how many level of indirection we can use but practically we seldom use morethan two.Each level of pointer indirection requires a separate indirection operator when it is dereferenced .

In the above figure to refer to ‘a’ using the pointer ‘p’, we have to dereference it as shown below.

*p

To refer to the variable ‘a’ using the pointer ‘q’ ,we have to dereference it twice toget to the integer ‘a’ because there are two levels of indirection(pointers) involved.If we dereference it only once we are referring ‘p’ which is a pointer to an integer .Another way to say this is that ‘q’ is a pointer to a pointer to an integer.The doule dereference is shown below:

**q

In above example all the three references in the printf statement refer to the variable ‘a’. The first printf statement prints the value of the variable ‘a’ directly,second uses the pointer ‘p’,third uses the pointer ‘q’.The result is the value 58 printed 3 times as below

```
58      58      58
```

DYNAMIC MEMORY ALLOCATION :

Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running. It effectively postpones the data definition ,but not the declaration to run time.

To use dynamic memory allocation ,we use either standard data types or derived types .To access data in dynamic memory we need pointers.

MEMORY ALLOCATION FUNCTIONS:

Four memory management functions are used with dynamic memory. Three of them,malloc,calloc,and realloc,are used for memory allocation. The fourth ,free is used to return memory when it is no longer needed. All the memory management functions are found in standard library file(stdlib.h).

BLOCK MEMORY ALLOCATION(MALLOC) :

The malloc function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a void pointer to the first byte of the allocated memory. The allocated memory is not initialized.

Declaration:

```
void *malloc (size_t size);
```

The type size_t is defined in several header files including Stdio.h. The type is usually an unsigned integer and by the standard it is guaranteed to be large enough to hold the maximum address of the computer. To provide portability the size specification in malloc's actual parameter is generally computed using the sizeof operator. For example if we want to allocate an integer in the heap we will write like this:

```
Pint=malloc(sizeof(int));
```

Malloc returns the address of the first byte in the memory space allocated. If it is not successful malloc returns null pointer. An attempt to allocate memory from heap when memory is insufficient is known as overflow.

The malloc function has one or more potential error if we call malloc with a zero size, the results are unpredictable. It may return a null pointer or it may return someother implementation dependant value.

Ex:

```
If(!(Pint=malloc(sizeof(int))))
        // no memory available
   exit(100);
        //memory available
…
```

In this example we are allocating one integer object. If the memory is allocated successfully,ptr contains a value. If does not there is no memory and we exit the program with error code 100.

CONTIGIOUS MEMORY ALLOCATION(calloc) :

Calloc is primarily used to allocate memory for arrys.It differs from malloc only in that it sets memory to null characters. The calloc function declaration:

```
Void *calloc(size_t element_count, size_t element_size);
```

The result is the same for both malloc and calloc.

calloc returns the address of the first byte in the memory space allocated. If it is not successful calloc returns null pointer.

Example:

```
If(!(ptr=(int*)calloc(200,sizeof(int))))
        //no memory available
   exit(100);
```

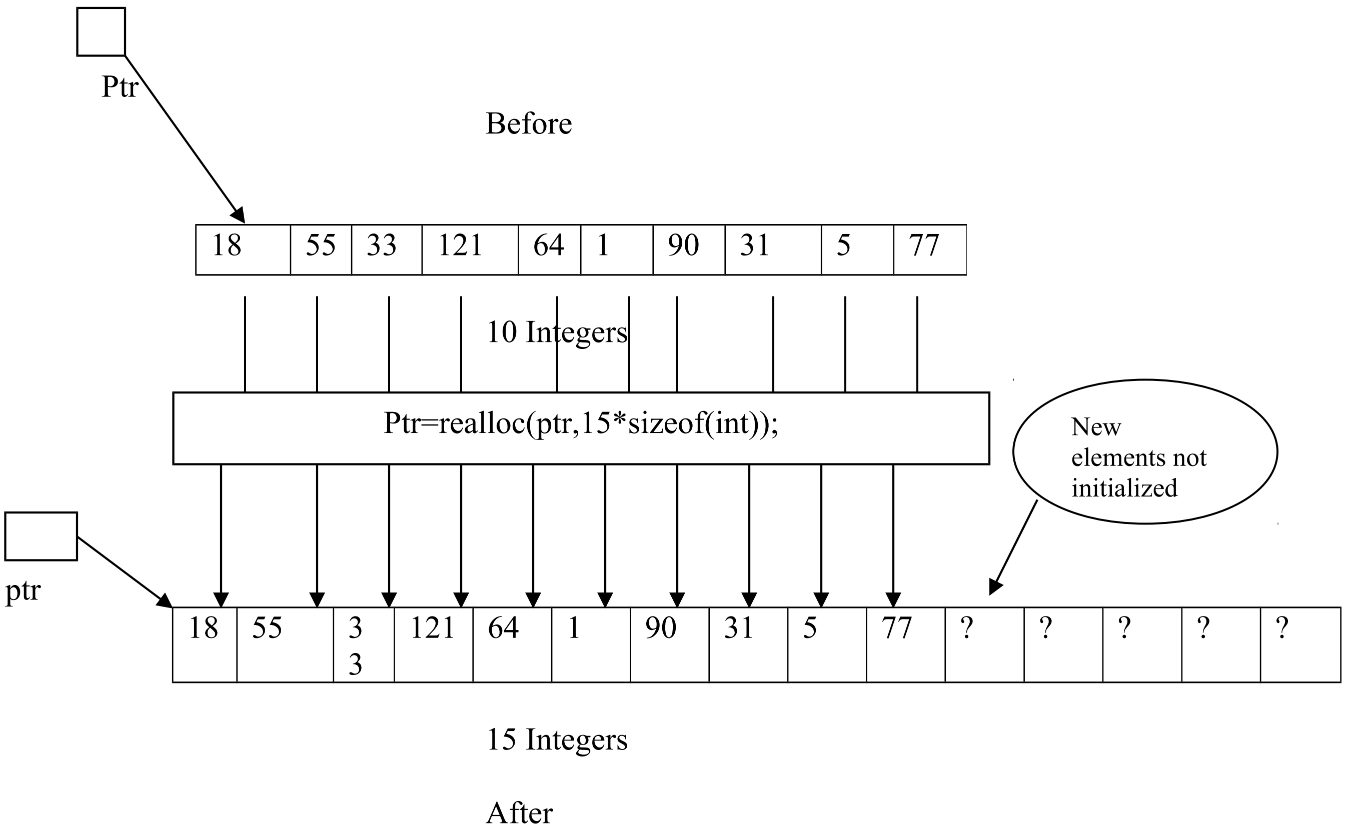
```
                        //memory available
                  …
```

In this example we allocate memory for an array of 200 integers.

REALLOCATION OF MEMORY(realloc):

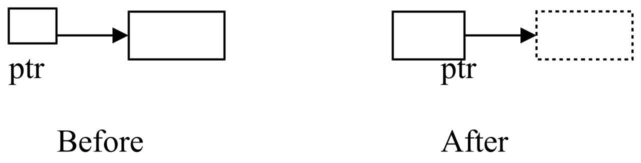
The realloc function can be highly inefficient and therefore should be used advisedly. When given a pointer to a previously allocated block of memory realloc changes the size of the block by deleting or extending the memory at the end of the block. If the memory can not be extended because of other allocations realloc allocates completely new block,copies the existing memory allocation to the new location,and deletes the old allocation.

```
Void *realloc(void*ptr,size_t newsize);
```



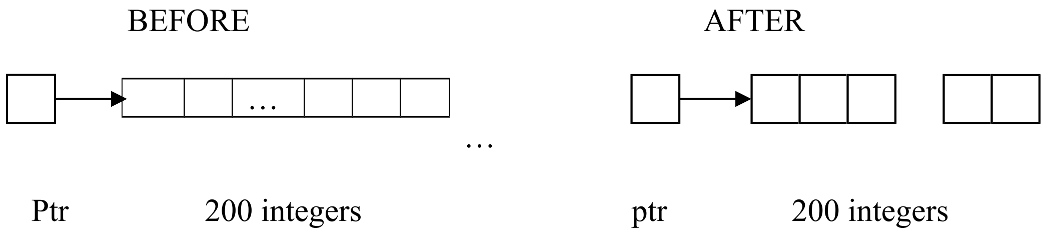
Releasing Memory(free):When memory locations allocated by malloc,calloc or realloc are no longer needed, they should be freed using the predefined function free. It is an error to free memory with a null pointer, a pointer to other than the first element of an allocated block, a pointer that is a different type then the pointer that allocated the memory, it is also a potential error to refer to memory after it has been released.

```
Void free(void *ptr);
```



Free(ptr);

In above example it releases a single element to allocated with a malloc,back to heap.



Free(ptr);

In the above example the 200 elements were allocated with calloc. When we free the pointer in this case, all 200 elements are return to heap. First, it is not the pointers that are being released but rather what they point to. Second , To release an array of memory that was allocated by calloc , we need only release the pointer once. It is an error to attempt to release each element individually.

Releasing memory does not change the value in a pointer. Still contains the address in the heap. It is a logic error to use the pointer after memory has been released.

Note: Memory space should be freed whenever it is not required. The operating system will release all memory when your program terminates.

COMMAND LINE ARGUMENTS:

When we see the main() always it has empty parantheses. We can also pass arguments to main(). The arguments can be passed from Operating System Command Prompt. Therefore they are called as Command Line Arguments.

We have two such arguments namely argc and argv. Argc is an integer variable where as argv is array of pointers to character, i.e., an array of strings. Each string in this array will represent a argument that is passed to main. The value of argc will indicate the number of arguments passed. The Skeleton of main along with arguments will look like.

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

}
```

In order to pass one or more arguments to the program when it is executed from the Operating System, the parameters must follow the program name on the Command line as.

```
Programme arg1,arg2………. argn
```

The individual parameters must be separated by a blank space.
The program name will be stored as the first item argv, followed by each of the parameters.

If there are n parameters, there will be (n+1) entries in argv. argc will automatically be assigned the value (n+1).

Eg:-

```
#include<stdio.h>
main(int argc, char *argv[])
{
        int count;
        printf("argc = %d\n",argc);
        for (count =0;count<argc;count++)
        printf("argv[%d] = %s\n", count, argv[count]);
}
```

Suppose if the above program is saved as cmdprg.c, then the command line for executing program is like

```
C:\>cmdprg one two three
Then output will be ------------
argc =4
argv[0] =cmdprg
argv[1] =one
argv[2] =two
argv[3] = three.
```

```
/* Demonstrate the use of command-line arguments    */

#include<stdio.h>
 #include<string.h>
#include<stdlib.h>
int main(int argc,char * argv[])
{
//statements
printf("the number of arguments: %d\n",argc);
printf("the name of the program : %s\n",argv[0]);
for(int i=1;I<argc; i++)
printf("user value no. %d: %s\n ", I,argv[i]);
return 0;
}//main
```

output:

```
c:>cmdline hello
The number of arguments :2
The name of the program:cmdline
User value number.1:hello
```

CHARACTER ARRAYS (STRINGS) :

A String is an array of characters. Any group of characters (except double quote sign)defined between double quotes is a constant string.
Ex: "C is a great programming   language".

If we want to include double quotes.

Ex: "\"C is great \" is norm of programmers ".
Declaring and initializing strings :-
A string variable is any valid C variable name and is always declared as an array.
    char string name [size];

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

String can be initialized when declared as

1. char city[g]= "NEW YORK';

2. char city[g]= {‘N’,’E’,’W’,’ ‘,’Y’,’O’,’R’,’K’,’/0’};

C also permits us to initializing a String without specifying size.

Ex:- char Strings[ ]= {‘G’,’O’,’O’,’D’,’\0’};

READING STRINGS FROM USER:

%s format with scanf can be used for reading String.

```
char address[15];
scanf(“%s”,address);
```

The problem with scanf function is that it terminates its input on first white space it finds. So scanf works fine as long as there are no spaces in between the text.

Reading a line of text :

If we are required to read a line of text we use getchar(). which reads a single characters. Repeatedly to read successive single characters from input and place in character array.

```
/* Program to read String using scanf & getchar */
#include<stdio.h>
main()
{
        char line[80],ano_line[80],character;
        int c;
        c=0;
        printf(“Enter String using scanf to read \n”);
        scanf(“%s”, line);
        printf(“Using getchar enter new line\n”);
        do
        {
                character = getchar();
                ano_line[c] = character;
                c++;
        } while(character !=’\n’);
        c=c-1;
        ano_line[c]=’\0’;
}
```

STRING INPUT/OUTPUT FUNCTIONS:

C provides two basic ways to read and write strings.First we can read and write strings with the formatted input/output functions,scanf/fscanf and prinf/fprinf.Second we can use a special set of strin only functions ,get string(gets/fgets)and put string(puts/fputs).

Formatted string Input/Output:

Formatted String Input:scanf/fscanf:

Declarations:

int fscanf(FILE ****stream*, const char ****format***, ...);**
int scanf(const char ****format*, ...);**
The ..scanf functions provide a means to input formatted information from a stream

fscanf	reads formatted input from a stream
scanf	reads formatted input from stdin

These functions take input in a manner that is specified by the format argument and store each input field into the following arguments in a left to right fashion.

Each input field is specified in the format string with a conversion specifier which specifies how the input is to be stored in the appropriate variable. Other characters in the format string specify characters that must be matched from the input, but are not stored in any of the following arguments. If the input does not match then the function stops scanning and returns. A whitespace character may match with any whitespace character (space, tab, carriage return, new line, vertical tab, or formfeed) or the next incompatible character.

INPUT CONVERSION SPECIFICATION:

FLAG:

There is only one flag for input formatting,The assignment suppression flag(*). The assignment suppression flag tells scanf that the next input field is to be read but not stored. It is discarded.

Example:

```
        Scanf(“%d%*c%f”, &x,&y);
```

WIDTH:

The width specifies the maximum width of the input(in characters).This allows us to break out a code that may be stored in the input without spaces.

SIZE:

A size specification is a modifier for the conversion code. Used in combination with the conversion code,specifies the type of the associative variable.

CONVERSION CODES:

Integer(d): The decimal (d) format code accepts a value from the input stream and formats it in to the specified variables. It reads only decimal digits and an optional plus or minus sign is the first character of the value.

Integer(i): The integer format code (i) allows the user to create decimal ,octal or hexadecimal numbers. Numbers starting with any digit other than zero are read and stored as decimal values. Numbers starting with zero are interpretes as octal values and are converted to decimal and stored. Hexadecimal numbers must be prefixed with 0x (or) 0X, The hexadecimal value is converted to decimal and stored.

Octal and hexadecimal(o,x): The octal (o) and hexadecimal (x) conversion codes perform unsigned conversion. For octal only valid input digits are 0…7. For Hexadecimal input valid digits are 0…9, a…f and A…F.

Scientific notation (e,g,a): The C languages uses three real format codes for scientific notation. In scientific notation the significand and exponent are specify separately. The significant part is a floating point number that contains as many significant digits as possible.

Count(n): To verify the number of input characters we use the n-conversion code. The code requires a matching variable address in to which scanf places the count of the characters input.

The Common Input Format Mistakes: Two common mistakes while formatting input stream, They are

Invalid Address
Datatype conflict

Invalid address: The common mistake is to use a data value rather than an address for the input parameter

Data type conflict: It is conflict between the format stream and the input stream. It occurs, For example when the format string calls for a numeric value and the input stream contains alphabetic character then scanf operation aborts.

Formatted String Output:printf/fprintf:

Declarations:

int fprintf(FILE ****stream*, const char ****format***, ...);**
int printf(const char ****format*, ...);**

The ..printf functions provide a means to output formatted information to a stream.

fprintf sends formatted output to a stream
printf sends formatted output to stdout
These functions take the format string specified by the *format* argument and apply each following argument to the format specifiers in the string in a left to right fashion. Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

Flags:

- Value is left justified (default is right justified). Overrides the 0 flag.
+ Forces the sign (+ or -) to always be shown. Default is to just show the - sign. Overrides the space flag.

Width:
The width of the field is specified here with a decimal value. If the value is not large enough to fill the width, then the rest of the field is padded with spaces (unless the 0 flag is specified). If the value overflows the width of the field, then the field is expanded to fit the value. If a ***** is used in place of the width specifer, then the next argument (which must be an **int** type) specifies the width of the field. Note: when using the ***** with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted.

Precision:
The precision begins with a dot (.) to distinguish itself from the width specifier. The precision can be given as a decimal value or as an asterisk (*****). If a ***** is used, then the next argument (which is an **int** type) specifies the precision. Note: when using the ***** with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted. Precision does not affect the c type.

String Input/Output

In addition to the Formatted string functions,C has two sets of string functions that read and write strings without reformatting any data.These functions convert text file lines to strings and strings to text file lines.

Line to String:

gets():

Declaration:

`char *gets(char *`*str*`);`

Reads a line from **`stdin`** and stores it into the string pointed to by *str*. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. The newline character is not copied to the string. A null character is appended to the end of the string.

On success a pointer to the string is returned. On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

fgets():

Declaration:

`char *fgets(char *`*str*`, int `*n*`, FILE *`*stream*`);`

Reads a line from the specified stream and stores it into the string pointed to by *str*. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. The newline character is copied to the string. A null character is appended to the end of the string.

On success a pointer to the string is returned. On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

String to Line:

puts:

Declaration:

`int puts(const char *`*str*`);`

Writes a string to **`stdout`** up to but not including the null character. A newline character is appended to the output.

On success a nonnegative value is returned. On error **`EOF`** is returned.

fputs():

Declaration:

int fputs(const char ****str*, FILE ****stream***);**
Writes a string to the specified stream up to but not including the null character.

On success a nonnegative value is returned. On error **EOF** is returned.

ARRAY OF STRINGS:

Arrays of strings (arrays of character arrays) can be declared and handled in a similar manner to that described for 2-D arrays.

Consider the following example:

```
#include< stdio.h>

void main(void)
{
char names[2][8] = {"Frans", "Coenen"};

/* Output */

printf("names = %s, %s\n",names[0],names[1]);
printf("names = %s\n",names);

/* Output initials */

printf("Initials = %c. %c.\n",names[0][0],names[1][0]);
}
```

Here we declare a 2-D character array comprising two "roes" and 8 "columns". We then initialise this array with two character strings. The output the array we need to index into each row --- using the 2-D array name on its own (strings) as a pointer cause only the first element ("row") to be produced. Note that we can still index to individual elements using index pairs. The output from the above will be:

```
names = Frans, Coenen
names = Frans
Initials = F. C.
```

ARRAYS OF STRINGS AS POINTER ARRAYS

The declaration of an array of character pointers is an extremely useful extension to single string pointer declarations.For example:

```
char *names[2];
```

Creates a two element array of character pointers (i.e. each element is a pointer to a character). As individual pointers each can be assigned to point to a string using a normal assignment statement:

```
names[0] = "Frans";
names[1] = "Coenen";
```

Alternatively we can initialise the names array as shown in the following programs:

```
#include< stdio.h>

void main(void)
{
char *names[2] = {"Frans", "Coenen"};

/* Output */

printf("names = %s, %s\n",names[0],names[1]);

/* New assignments */

names[0] = "Ray";
names[1] = "Paton";

/* Output */

printf("names = %s, %s\n",names[0],names[1]);
}
```

In the above the declaration of names both creates an array of pointers and initilises the pointers with appropriate addresses. Once addresses have been assigned to the pointers, each pointer can be used to access its corresponding string. The output from the above is as follows:

```
names = Frans, Coenen
names = Ray, Paton
```

STRING/DATA CONVERSION:

String to Data Conversion:

sscanf():

The string scan function is called sscanf. The sscanf function accepts a string from which to read input, then, in a manner similar to `printf` and related functions, it accepts a template string and a series of related arguments. It tries to match the template string to the string from which it is reading input, using conversion specifier like those of `printf`.

The sscanf function is just like the deprecated parent `scanf` function, except that the first argument of sscanf specifies a string from which to read, whereas `scanf` can only read from standard input. Reaching the end of the string is treated as an end-of-file condition.

```
    ret = sscanf( s, format [, p1, p2, ...] );
```

const char *s;
points to the string which is to be read.
const char *format;
is a format string similar to that for "scanf".
p1, p2, p3, ...
are pointers to various data types.
int ret;
is the number of matching data items that were read in. This may be zero if the first data item found does not match the type that was expected. If an error or end of string occurs before the first item could be read in, EOF is returned.
Description:

"sscanf" reads input from the string "s" and assigns it to the areas of memory pointed to by the pointers "p1", "p2", and so on. The "format" string indicates how to interpret the characters being read in.

Important difference between sscanf and printf is that the arguments to sscanf must be pointers; this allows sscanf to return values in the variables they point to. If you forget to pass pointers to sscanf, you may receive some strange errors, and it is easy to forget to do so; therefore, this is one of the first things you should check if code containing a call to sscanf begins to go awry.

A sscanf template string can contain any number of any number of whitespace characters, any number of ordinary, non-whitespace characters, and any number of conversion specifiers starting with %. A whitespace character in the template string matches zero or more whitespace characters in the input string. Ordinary, non-whitespace characters must correspond exactly in the template string and the input stream; otherwise, a matching error occurs. Thus, the template string " foo " matches "foo" and " foo ", but not " food ".

Data to String Conversion:

The string print function,sprintf. Returns a string formatted by the usual printf() conventions of the C library function sprintf().

```
   i = sprintf( s, control [, arg1, arg2, ...] );
```

char *s;
    is the string to which the output should be written.
const char *control;
    is a "printf" control string.
arg1, arg2, ...
    are the values to be output.
int i;
    is the number of characters that were output. If a write error occurred, a negative number is returned.

Description:

"sprintf" writes to the string "s". "sprintf" converts, formats, and prints its arguments under control of the string "control.

STRING HANDLING FUNCTIONS:

strcat( )	Concatenates two Strings
strcmp( )	Compares two Strings
strcpy( )	Copies one String Over another
strlen( )	Finds length of String

strcat() :

This function yours two strings together.
strcat(string1,string2);
string1 = VERY

string2 = FOOLISH

strcat(string1,string2);

string1=VERY FOOLISH
string2 = FOOLISH

strcmp() function :

This function compares two strings identified by arguments and has a value 0 if they are equal. If they are not, it has the numeric difference between the first non-matching characters in the Strings.

```
strcmp(string1,string2);
Ex:-    strcmp(name1,name2);
        strcmp(name1,"John");
        strcmp("ROM","Ram");
```

strcpy() function :

It works almost as a string assignment operators. It takes the form

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

string2 can be array or a constant.

strlen() function :

Counts and returns the number of characters in a string.

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

n→ integer variable which receives the value of length of string.

```
/* Illustration of string-handling */

#include<stdio.h>
#include<string.h>
main()
{
        char s1[20],s2[20],s3[20];
        int  X,L1,L2,L3;
        printf("Enter two string constants\n");
        scanf("%s %s",s1,s2);
        X=strcmp(s1,s2);
        if (X!=0)
        {
                printf("Strings are not equal\n");
                strcat(s1,s2);
        }
        else
        printf("Strings are equal \n");
        strcpy(s3,s1);
        L1=strlen(s1);

        L2=strlen(s2);
        L3=strlen(s3);
        printf("s1=%s\t length=%d chars \n",s1,L1);
        printf("s2=%s\t length=%d chars \n",s2,L2);
        printf("s3=%s\t length=%d chars \n",s3,L3);
}
```

IMPORTANT QUESTIONS UNIT - IV

1. (a) What is a pointer? How is a pointer initiated? Give an example?

   (b) State whether each of the following statement is true or false. Give reasons.

      (i) An Integer can be added to a pointer.

(ii) A pointer can never be subtracted from another pointer.

(iii) When an array is passed as an argument to a function, a pointer is passed.

(iv) Pointers can not be used as formal parameters in headers to function definitions.

(c) If m and n have been declared as integers and p1 and p2 as pointers to integers, then find out the errors, if any, in the following statements.

(i) p1=&m;

(ii) P2=n;

(iii) M=p2-p1;

(iv) *p1=&n;

2. (a) Explain the process of accessing a variable through its pointer. Give an Example.

(b) Write a C program using pointers to read in an array of integers and print its elements in reverse order?

3. (a) Write a C program using pointer for string comparison.

(b) Write a C program to arrange the given numbers in ascending order using pointers.

4. (a) How to use pointer variables in expressions? Explain through examples?

(b) Write a C program to illustrate the use of pointers in arithmetic operations.

5. (a) Explain the process of declaring and initializing pointers. Give an example.

(b) Write a C program that uses a pointer as a function argument.

6. Write short notes on pointers.

7. (a) Write a C program to illustrate the use of structure pointer.

(b) Explain the effects of the following statements.

(i) int a, *b=&a;

(ii) int p, *p;

(iii) char *s;

(iv) A=(float*)&X;

8. (a) Write a C program to compute the sum of all elements stored in an array using pointers.

(b) Write a C program using pointers to determine the length of a character string.

9. (a) In what way array is different from an ordinary variable?

(b) What conditions must be satisfied by the entire elements of any given array?

(c) What are subscripts ? How are they written? What restrictions apply to the values that can be assigned to subscripts?

(d) What advantages are there in defining an array size in terms of a symbolic constant rather than as a fixed integer quantity?

(e) Write a program to find the largest element in an array.

FILL IN BLANKS:-

1.Strings is an array of _____________________

2.The format for reading strings in a scanf statement is ____________________.

3.strcat( ) is used for ___________________ of two strings.

4.strcmp( ) is used for __________________ of two strings.

5.For copying one string over another ______________function is used.

6.To determine the length of a string we use _______________ function.

7. strcat ( )Concatenates ___________________ string with another.

8. In strlen( ) the counting ends at first ____________________ character.

9. printf and scanf belong to the category of ___________ functions.

10. ________________ functions are developed by the user at time of writing functions.

11. Every program must have _______________ function to indicate where the program has to begin its execution.

12. There are _____________ category of functions.

13. Functions are categorized depending on __________________ and their _____________ values.

14. A function by default returns ______________________ data type.

15. A function can be forced to return another datatype (other than default) by specifying the _________ before the function header.
16. ________________ is a process in which a function calls itself.
17. The ______________ of variable determines over what parts of the program a variable is actually available for use.
18. Longevity refers to period during which a variable returns a given value during ____________ of a program.
19. The variables may be broadly categorized as _________________ or __________________.
20. The four storage classes are Automatic, External, _______________ and _____________variables.
21. By default, storage is __________________.

True or false:-

1. Automatic Storage is by default.
2. Register access is faster than other storage.
3. User defined functions are developed by user.
4. Arrays can be of variable data type.
5. A Single array can have elements of different data types.
6. String is an array.
7. %s format is used for string declaration in scanf & printf.
8. %A format is used for arrays.
9. Recursion is a self calling process of functions.
10. Functions can be Nested.
11. A function can take values (arguments) and return values.
12. String 3 = String1 + String2
13. A null character is present at the end of character String.
14. Arrays are primary data types.
15. Strings.h is a header file containing string handling functions like strlen( ), etc.

16. Is following declaration correct.
    (i) char city[ ] = {“John”}
    (ii) char city[ ] =”John”;
17. There are two types of functions. User-defined and library functions.
18. Using user defined functions is not a good programming practice.
19. To call a function you just write a function name.
20. To return a value to the function i.e., calling function you use the Keyword return.

UNIT - V

STRUCTURES :

A Structure is a collection of elements of dissimilar data types. Structures provide the ability to create user defined data types and also to represent real world data.

Suppose if we want to store the information about a book, we need to store its name (String), its price (float) and number of pages in it(int). We have to store the above three items as a group then we can use a structure variable which collectively store the information as a book.

Structures can be used to store the real world data like employee, student, person etc.

Declaration :

The declaration of the Structure starts with a Key Word called Struct and ends with ; . The Structure elements can be any built in types.

```
struct <Structure name>
{
        Structure element 1;
        Structure element 2;
                -
                -
                -
        Structure element n;
};
```

Then the Structure Variables are declared as

```
        struct < Structure name > <Var1,Var2>;

        Eg:-    struct emp
                {
                        int empno.
                        char ename[20];
                        float sal;
                };
        struct emp  e1,e2,e3;
```

The above Structure can also be declared as :

```
struct emp                                    struct
{                                             {
        int empno;                            int empno;
        char ename[20];          (or)         char ename[20];
        float sal;                            float sal;
}e1,e2,e3;                                    }e1,e2,e3;
```

Initialization :

Structure Variables can also be initialised where they are declared.

```
struct emp
{
        int empno;
        char ename[20];
        float sal;
};
struct emp e1 = { 123,"Kumar",5000.00};
```

To access the Structure elements we use the .(dot) operator.
To refer empno we should use e1.empno
To refer sal we whould use e1.sal

Structure elements are stored in contiguous memory locations as shown below. The above Structure occupies totally 26 bytes.

e1.empno	E1.ename	E1.sal
123	Kumar	5000.00
2000	2002	2022

1. Program to illustrate the usage of a Structure.

```
main()
{
    struct emp
    {
```

```
                int empno;
                char ename[20];
                float sal;
            };
            struct emp e;
            printf ("  Enter Employee number: \n");
            scanf("%d",&e.empno);
            printf ("  Enter Employee Name: \n");
            scanf("%s",&e.empname);
            printf ("  Enter the Salary: \n");
            scanf("%f",&e.sal);
            printf (" Employee No = %d", e.empno);
            printf ("\n Emp Name = %s", e.empname);
            printf ("\n Salary  = %f", e.sal);
        }
```

NESTING OF STRUCTURES :

One Structure can be enclosed in another Structure. The Structure which is to be nested must be declared before it is used. We can nest a Structure within a Structure, which is in still another Structure and so on. Nesting of Structure can be done to any no of levels.

The elements of the nested Structure are accessed by using one more dot(.) operator.

Eg:-

```
struct date
{
        int dd, mm, yy;
};
struct student
{
        int rno;
        char sname[15];
        struct date dob;
};
struct student s;
```

Here, dob is the Nested Structure. To access this we use s.dob.dd or s.dob.mm or s.dob.yy.

/* Program to read Student Details and Calculate total and average using structures */

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

```
{
        struct stud
        {
                int rno;
                char sname[20];
                int m1,m2,m3;
        };
        struct stud s;
        int tot;
        float avg;

        printf("Enter the student roll number: \n");
        scanf("%d",&s.rno);
        printf("Enter the student name: \n");
        scanf("%s",&s.sname);
        printf("Enter the three subjects marks: \n");
        scanf("%d%d%d",&s.m1,&s.m2,&s.m3);

        tot = s.m1 + s.m2 +s.m3;
        avg = tot/3.0;



        printf("Roll Number : %d\n",s.rno);
        printf("Student Name: %s\n",s.sname);
        printf("Total Marks : %d\n",tot);
        printf("Average : %f\n",avg);
}
```

/* Program to read Item Details and Calculate Total Amount of Items*/

```
#include<stdio.h>
main()
{
        struct item
        {
                int itemno;
                char itemname[20];
                float rate;
                int qty;
        };
        struct item i;
        float tot_amt;
        printf("Enter the Item Number \n");
        scanf("%d",&i.itemno);
        printf("Enter the Item Name \n");
        scanf("%s",&i.itemname);
        printf("Enter the Rate of the Item \n");
        scanf("%f",&i.rate);
        printf("Enter the number of %s purchased ",i.itemname);
        scanf("%d",&i.qty);

        tot_amt = i.rate * i.qty;

        printf("Item Number: %d\n",i.itemno);
        printf("Item Name: %s\n",i.itemname);
        printf("Rate: %f\n",i.rate);
        printf("Number of Items: %d\n",i.qty);
        printf("Total Amount: %f",tot_amt);
}
```

/*Program to illustrate the use of Nested Structures */

```
#include<stdio.h>
main()
{
        struct date
        {
                int dd, mm, yy;
        };
        struct stud
        {
                int rno;
                char sname[15];
                struct date dob;
                int tot;
        };
        struct stud s;
        printf("Enter the student roll number: \n");
        scanf("%d",&s.rno);
        printf("Enter the student name: \n");
        scanf("%s",&s.sname);
        printf("Enter the date of birth of student(dd/mm/yy): \n");
        scanf("%d%d%d",&s.dob.dd,&s.dob.mm,&s.dob.yy);

        printf("Roll Number : %d\n",s.rno);
        printf("Student Name: %s\n",s.sname);
        printf("Date of Birth: %d/%d/%d\n", s.dob.dd,s.dob.mm,s.dob.yy);
        printf("Total Marks : %f\n",s.tot);
}
```

ARRAYS OF STRUCTURES :

To store more number of Structures, we can use array of Structures. In array of Structures all elements of the array are stored in adjacent memory location.

/* Program to illustrate the usage of array of Structures.

```
main()
{
        struct book
        {
                char name[20];
                float price;
                int pages;
        };
        struct book b[10];
        int i;
        for (i=0;i<10;i++)
        {
                print("\n Enter Name, Price and Pages");
                scanf("%s%f%d", b[i].name,&b[i].price,&b[i].pages);
        }
        for (i i=0;i<10;i++)
                printf(" \n%s%f%d", b[i].name,b[i].price,b[i].pages);
}
```

STRUCTURES WITH FUNCTIONS :

The entire Structure can be passed to a function at once. Whenever a Structure is passed to a function the Structure should be declared outside the main(), otherwise it will not be known to all the functions in the program.

Program to illustrate the use of Structures with functions.

```
struct stud
{
        int rno;
        char name[20];
        int tot;
};
```

```
main()
{
        struct stud s = { 111, “Ramoo”,500};
        display(s);
}
display(struct stud s)
{
        printf(“Roll Number: %d\n”, s.rno);
        printf(“Student Name: %s\n”, s.name);
        printf(“Total Marks: %d\n”, s.tot);
}
```

STRUCTURE POINTER :

A Pointer pointing to a struct is called Structure Poniter. A Structure Pointer can be used to store the address of a Structure Variable. An arrow operator (→) is used to refer the Structure elements when a pointer is used with a Structure.

/* Program to illustrate the usage of Structures with Pointer */

```
struct emp
{
        int empno;
        char ename[20];
        float sal;
};

main()
{
        struct emp e;

        reademp(&e);
        displayemp(&e);
}
reademp(struct emp *e)
{
        printf(“Enter Emp Details:”);
        scanf(“\n%d%s%f”, &e→empno, e→ename, &e→sal);
}
display emp(struct emp *e)
{
        printf("empno:%d\tename:%s\tSalary=%f",e→empno, →ename,e→sal);
}
```

SELF REFERENTIAL STRUCTURES:

A Self referential Structure is one that includes with in it's structure atleast one member which is a pointer to the same structure type. With self referential structures we can create very useful data structures such as linked lists,trees etc.

Example:

```
Struct tag
{
char name[40];
struct emp*next;
};
```

A structure contains teo members a 40 element character array called name,and a pointer to another structure of the same called next.

APPLICATIONS OF STRUCTURES :

Structures can be used for a Variety of Applications like:

a) Database Management.
   (To maintain data about employees in an organisation, book in a library, items in a Store etc)
b) Changing the size of the Cursor.
c) Clearing the contents of the Screen.
d) Placing the Cursor at appropriate position on Screen.
e) Drawing any graphics Shape on the Screen.
f) Receiving a Key from the Keyboard.
g) Checking the memory size of the Computer.
h) Finding the list of equipment attached to the Computer.
i) Formatting a Floppy.
j) Hiding a file from the Directory.
k) Displaying the Directory of a Disk.
l) Sending the Output to Printer.
m) Interacting with the Mouse.

UNIONS :

Union, similar to Structures, are collection of elements of different data types. However, the members within a union all share the same storage area within the computer's memory, whereas each member within a Structure is assigned its own unique Storage area.

Structure enables us to treat a member of different variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables. That is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.

Unions are used to conserve memory. Unions are useful for applications involving multiple members, where values need not be assigned to all of the members at any given time. An attempt to access the wrong type of information will produce meaningless results.

The union elements are accessed exactly the same way in which the structure elements are accessed using dot(.) operator.

The difference between the structure and union in the memory representation is as follows.

```
struct cx
{
        int i;
        char ch[2];
};
struct cx s1;
```

The Variable s1 will occupy 4 bytes in memory and is represented as

← ---------- s.i -------→		← s.ch[0]→	← s.ch[1]→
4002	4003	4004	4005

The above datatype, if declared as union then

```
union ex
{
        int i;
        char ch[2];
}
union ex u;
```

←--------u.i ---------------→

/* Program to illustrate the use of unions */

```
main()
{
        union example
        {
                int i;
                char ch[2];
        };
        union exaple u;

        u.i = 412;

        print("\n u.i = %d",u.i);
        print("\n u.ch[0] = %d",u.ch[0]);
        print("\n u.ch[1] = %d",u.ch[1]);

        u.ch[0] = 50; /* Assign a new value */

        print("\n\n u.i = %d",u.i);
        print("\n u.ch[0] = %d",u.ch[0]);
        print("\n u.ch[1] = %d",u.ch[1]);
}
```

Output :

```
        u.i  = 512
        u.ch[0] = 0
        u.ch[1] =2
        u.i  = 562
        u.ch[0] = 50
        u.ch[1] =2
```

A union may be a member of a structure, and a structure may be a member of a union. And also structures and unions may be freely mixed with arrays.

/* Program to illustrate union with in a Structure */

```
main()
{

        union id
        {

                char color;
                int size;

        };

        struct {

                char supplier[20];
                float cost;
                union id desc;

        }pant, shirt;

        printf("%d\n", sizeof(union id));

        shirt.desc.color = 'w';
        printf("%c %d\n", shirt.desc.color, shirt.desc.size);

        shirt.desc.size = 12;
        printf("%c %d\n", shirt.desc.color, shirt.desc.size);

}
```

/* Program to raise a number to a power */

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

        typedef union {

                float fexp;      /* float exponent */
                int nexp;        /* integer exponent */

        }nvals;

        typedef struct{
```

```
        float x;            /* value to raised to a power */
        char flag;          /* ‘f’ if exponent is float ‘i’ if exponent is integer */

        nvals exp;          /* union containing exponent */

    }values;

float power (values a )         /* function calculations power */
{
    int i;
    float y = a.x;
    if (a.flag = = ‘i’){        /* integer exponent */

        if (a.exp.nexp = = 0)   /* zero exponent */
            y = 1.0;
        else {
            for (i = 1;i<abs(a.exp.nexp);++i)
                y * = a.x;
            if (a.exp.nexp<0)
                y = 1.0/y;          /* negative integer exponent */
        }
    }
    else
        y = exp(a.exp.fexp*log(a.x));    /* floating point exponent */
    return(y);
}

main()
{

    values a ;
    int i;
    float n,y;

    printf(“Enter the value for X”);
    scanf(“%f”,&a.x);

    printf(“Enter the value for n”);
    scanf(“%f”,&n);

        /* determine type of exponent */

    i = (int)n;
    a.flag = (i = = n) ? ‘i’ : ‘f’ 3
```

```
          if((a.flag = =  ‘i’)

                    a.exp.nexp = i;

          else

                    a.exp.fexp = n;

                    /* raise X to the power */

          if(a.flag = = ‘f’ && a.x <= 0.0)

                    printf(“ERROR, cannont raise negative number to floating point power”);

          else{

                    y = power(a);
                    printf(“\n y = % .4f”,y);

          }
}
```

TYPEDEF:

Typedef is used to create new datatype.The statement typedef is used while defining the new data type.

Syntax:

```
          Typedef type dataname;
```

Here , type is the data type and dataname is the user defined name for that type.

```
          Typedef int hours;
```

Here, hours is another name for int and now we can use hours instead of int in the program as follows:

Program:

```
#define H 60
main()
{
typedef int hours;
hours hrs;
clrscr();
printf("Enter hours\n");
scanf("%d",&hrs);
printf("minutes=%d",hrs*H);
printf("\nseconds=%d",hrs*H*H);
}
```

output:

```
Enter hours: 2
Minutes=120
Seconds=7200
```

Explanation of program:

In the above example with typedef we had declared hours as an integer data type. Immediately after the typedef statement hrs is a variable of hours datatype which is similar to int. further the program calculates minuter and seconds using hrs variable.

BITFIELDS:

A bitfield allows structures to be subdivided into groups of bits. This process allows multiple variables to be packed into a single block of memory.

syntax :type_name <variable_name> : number_of_bits

The type can be char, short, int, or int64 (unsigned or signed) or any of the equivalent types. If the variable name is omitted, the given number of bits will be skipped.

For example:

```
int alpha : 5;
int       : 12;
  int beta  : 15;
```

would pack alpha and beta into one 32-bit value, but skip 12 bits in the middle.

Example: Suppose we want to store the following data about an employee. Each employee can:

(a) be male or female
(b) be single,married,divorced or widowed
(c) have one of the eight different hobbies
(d) can chose from any of the fifteen different schemes proposed by the company to pursue his/her hobby.

This means one bit is needed to store gender, two bits to store marital status, three for hobby and four for scheme i.e there is a need for ten bits altogether which means we can pack all this information in to a single integer since an integer is 16 bits long.
To do this using bit fields we declare the following structure:

```
Struct employee
{
unsigned gender:1;
unsigned mat_stat:2;
unsigned hobby:3;
unsigned scheme:4;
};
```

The colon in the above declaration tells the compiler that we are talking about bit fields and the number after it tells how many bits to allot for the field. Once we have established a bit field, we can reference it just like any other structure element as shown in the program given below:

```
#include<stdio.h>
#define MALE 0;
#define FEMALE 1;
#define SINGLE 0;
#define MARRIED 1;
#define DIVORCED 2;
#define WIDOWED 3;

void main()
{
        struct employee
        {
        unsigned gender:1;
        unsigned mar_stat:2;
        unsigned hobby:3;
        unsigned scheme:4;
        };
struct employee e;
e.gender=MALE;
e.mar_status=DIVORCED;
e.hobby=5;
e.scheme=9;

printf("\nGender=%d",e.gender);
printf("\nMarital status=%d",e.mar_status);
printf("\nBytes occupied by e=%d", sizeof(e));
}
```

output:

```
        Gender=0
        Marital status=2
        Bytes occupied by e=2
```

Two special bitfield modes that determine how the bits are packed into variables is:

1.padded bitfields

2. unpadded bitfields.

PADDED BITFIELDS:

With padded bitfields (the default), how the bits are packed into a variable depends upon the current endianness. By default, bits are packed right-to-left for little endian files, and left-to-right for big endian files. For example, for the bitfields:

```
      ushort a : 4;
      ushort b : 7;
        ushort c : 5;
```

In little endian mode, this structure would be stored as the bits:

```
      cccccbbb bbbbaaaa
```

(and stored on disk as bbbbaaaa cccccbbb). In big endian mode, this structure would be stored as the bits:

```
      aaaabbbb bbbccccc
```

(and stored on disk as aaaabbbb bbbccccc). Whether the bits are packed left-to-right or right-to-left can be controlled by the functions BitfieldLeftToRight, and BitfieldRightToLeft.

In this mode, the program will automatically add padding bits when needed. If the size of the type being defined changes, padding will be added so the bitfield will be defined on the next variable boundary. Also, if the specified bitfield would step across the boundary of a variable, padding is added so the bitfield starts at the next variable.

For example:

```
int  apple  : 10;
int  orange : 20;
int  banana : 10;
int  peach  : 12;
short grape : 4;
```

The bitfields apple and orange would be packed into one 32 bit value. However, banana steps over the variable boundary, so 2 bits of padding are added so that it starts at the next 32 bit value. Banana and peach are packed into another 32-bit value, but because the size of the type changes with grape, an extra 10 bits of padding is added before grape is defined.

UNPADDED BITFIELDS:

Unpadded bitfield mode treats the file as one long bit stream. No padding bits are added if the variable type changes or if the bits cannot be packed into a single variable. The unpadded mode can be entered by calling the function BitfieldDisablePadding (padding can be turned back on by calling BitfieldEnablePadding).

In unpadded bitfield mode, each variable defined reads some bits from the bitstream. For example:

```
BitfieldDisablePadding();
short a : 10;
int   b : 20;
short c : 10;
```

Here a reads the first 10 bits from the file, b reads the next 20 bits from the file, and so on. If the bitfields are defined as reading from right to left (this is the default for little endian data and can enabled using the function BitfieldRightToLeft), the variables would be stored as the bits:

aaaaaaaa bbbbbbaa bbbbbbbb ccbbbbbb cccccccc

If the bitfields are defined as reading from left to right (this is the default for big endian data and can enabled using the function BitfieldLeftToRight), the variables would be stored as the bits:

```
aaaaaaaa aabbbbbb bbbbbbbb bbbbbbcc cccccccc
```

NOTE: When declaring structures containing unpadded bitfields, no extra padding bits are added between the structures. (Note that internally, any unpadded right-to-left bitfield is forced to be declared in little endian mode and any unpadded left-to-right bitfield is forced to be declared in big endian mode.)

ENUMERATED DATA TYPES

In computer programming, 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.

Enumerated data type variables can only assume values which have been previously declared.

```
enum month { jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
enum month this_month;

this_month = feb;
```

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *this_month* is declared to be of the same type as month, then is assigned the value associated with feb. This_month cannot be assigned any values outside those specified in the initialization list for the declaration of month.

```
#include <stdio.h>

        main()
        {
                char *pwest = "west",*pnorth = "north", *peast="east",
*psouth="south";
                enum location { east=1, west=2, south=3, north=4};
```

```
          enum location direction;

          direction = east;

          if( direction == east )
                    printf("Cannot go %s\n", peast);
}
```

The variables defined in the enumerated variable *location* should be assigned initial values.

IMPORTANT QUESTIONS UNIT - V

1. (a) What are Bitfields? What are its advantages? What is its syntax?

   (b) Write a C program to store the information of vehicles. Use bit fields to store the status information. Assume the vehicle object consists of type, fuel and model member fields. Assume appropriate number of bits for each field.

2. Write a C program to compute the monthly pay of 100 employees using each employee's name, basic-pay. The DA is computed as 52% of the basic pay. (Gross-salary =Basic-pay+DA). Print the employee's name and gross salary.

3. (a) Write a C program to illustrate the comparison of structure variables.

   (b) What is the use of structure? Given an example for a structure with initialized values.

4. (a) Describe nested structures. Draw diagrams to explain nested structure.

   (b) Write a program to declare pointers as members of the structure and display the contents of that structure. Define a structure object boy with three fields: name, age and height.

5. (a) Differentiate between a structure and union with respect to allocation of memory by the compiler. Give an example of each.

   (b) Write a program to read n records of students and find out how many of them have passed. The fields are student's roll-no, name, marks and result. Evaluate the result as follows.

   If marks>35 then
   Result=”pass” else “Fail”

6. (a) Explain with an example how the structure can be organized in the C language.

   (b) Write a C program to print maximum marks in each subject along with the name of the student by using structures. Take 3 subjects and 3 student's records.

7. (a) Distinguish between an array of structures and an array with in a structure. Give an example of each.

(b) Write a C program using structure to create a library catalogue with the following fields: access number, author's name, title of the book, year of publication, publisher's name, price.

8. Write a program using structures to display the following information for each customer: name, account number, street, city, old balance, current payment, new balance, account status.

9. Define structure and write the general format for declaring and accessing members.

10. Consider a structure master that includes the information like name ,code, pay, experience. Write a program to delete and display the information contained in master variables for a given code.

11. (a) Write a C program to illustrate the comparison of structure variables.

(b) What is the use of a structure? Give an example for a structure with initialized values.

12. Define a structure to represent a date. Use your structures that accepts 2 different dates in the format mmdd of the same year and do the following: Write a C program to display the month names of both dates.

13. (a) Explain the advantages of structure type over the array type variable.

(b) Define a structure that represents a complex number(contains 2 floating point members, called real and imaginary).Write a C program to add, subtract, multiply two complex numbers.

15. (a) How to compare structure variables? Give an example.

(b) Define a structure type *struct ABS* ,that contains name, age, designation and salary. Using this structure, write a C program to read this information for one person from the keyboard and print the same on the screen.

16. (a) How are structure elements accessed using pointer? Which operator is used? Give an example.

(b) Write a program to use structure with in union. Display the contents of structure elements.

17. (a) What is the use of struct keyword? Explain the use of dot operator. Give an example for each.

(b) Write a C program to accept records of different states using array of structures. The structure should contain state, population, literacy rate and income. Display the state whose literacy rate is highest and whose income is highest.

UNIT - VI

FILES :

Files are used to store the information permanently and to access information whenever necessary. Files are stored in the Secondary Storage Devices like Floppy Disks, Hard Disks etc. Unlike other programming languages, C does not distinguish between sequential and direct access (random access) files.

There are two different types of files in C. They are Stream – Oriented (or Standard) files and System-Oriented (or Low-Level) files.

STREAMS:

All input and output operations in C is performed with streams. Stream is a sequence of characters organized into lines (text stream), each line consists of 0 or more characters and ends with the newline character '\n'. Streams provide communications channels between files and programs. A stream can be connected to a file by opening it and the connection is broken by closing the stream.

When program execution begins, three files and their associated streams are connected to the program automatically. They are

- the standard input stream
- the standard output stream
- the standard error stream

Standard input: It is connected to the keyboard & it enables a program to read data from the keyboard.
Standard output: It is connected to the screen & it enables a program to write data to the screen.
Standard error: It is+ connected to the screen & all error messages are output to standard error.

Operating systems often allow these streams to be redirected to other devices, Standard input, standard output and standard error are manipulated with file pointers **stdin**, **stdout** and **stderr.**

BINARY FILES:

Binary files contain fixed length records, They are in binary format (machine readable) not a human readable format like text files, Binary files can be accessed directly (i.e. random access). The record structure for a binary file is created using structures, They are more efficient than text files as conversion from ASCII to binary (reading) and vice versa for writing does not have to occur , They cannot be read easily by other non-C programs. Binary files are appropriate for online transaction processing systems.

E.x: Airline reservation, Order processing, Banking systems.

fwrite function:

The function fwrite is used to write to a binary file.

SYNTAX:

```
size_t fwrite (void *ptr, size_t size, size_t n, FILE *fptr);
```

- fwrite writes from array ptr, n objects of size size to file pointed to by fptr
- fwrite returns the number of objects written
- which is less than n if an error occurs

fread:

The function fread is used to read from a binary file.

SYNTAX:

```
size_t fread (void *ptr, size_t size,size_t n, FILE *fptr);
```

● fread reads n objects of size size from a file pointed to by fptr, and places them in array ptr
● fread returns the number of objects read which may be less than the number requested
● call to feof() is necessary to distinguish EOF and error condition.

TEXT FILES :

Text files are classified in to two types.These are:

1.Formatted Text files
2.Unformatted textfiles

FORMATTED TEXT FILES :

Formatted Text files contain variable length records must be accessed sequentially, processing all records from the start of file to access a particular record.

For formatted input and output to files we use fprintf and fscanf functions. These functions are the same as **printf** and **scanf** except that the output is directed to the stream accessed by the file pointer, **fptr.**

fprintf :

Fprintf is used to write a record to the file.

Syntax : int fprintf (FILE *fptr, const char* fmt,...)

Ex :fprintf (fptr,"%d %s %.2f\n",accNo,name,balance);

fprintf returns the number of characters written or negative if an error occurs.

fscanf :

Fscanf is used toRead a record from the file.

Syntax : int fscanf (FILE *fptr, const char* fmt, ...)

Ex :fscanf (fptr,"%d %20s %f",&accNo,name, &balance);

fscanf returns the number of input items assigned or EOF if an error or EOF occurs before any characters are input.Rewind(FILE* ptr) - resets the current file position to the start of the file.

SEARCHING THE FILE:

Accessing a particular record(s) requires

- a sequential read from the start of the file
- testing each record to see is it being searched for
- reset the file pointer to the start of the file ready for the next search.

UPDATING RECORDS:

Records cannot be updated , To update a text file the entire file is rewritten.

UNFORMATTED TEXT FILES :

stdio.h provides many functions for reading and writing to unformatted text files.

fgetc :

This function is used to read the next character from the stream,This is similar to getchar().

Syntax : int fgetc (FILE *stream)

It returns the character (as an integer) or **EOF** if end of file or an error occurs.

fputc :

This function is used to write the character **c** to the stream.This is similar to putchar(int).

Syntax : int fputc (int c, FILE *stream)

It returns the character or **EOF** for error.

fgets :

This function Reads at most the next n-1 characters from the stream into the array s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A '\0' is stored after the last character in the buffer. This is similar to gets(char*).

Syntax : char* fgets (char *s,int n,FILE *stream)

It returns s or NULL if EOF or error occurs

fputs :

This is used to write the string **s** to the stream. This is similar to puts(const char*).

Syntax : int fputs (const char *s, FILE *stream)

It returns EOF for error.

These file handling functions can be used instead of reading and writing characters and strings from the keyboard and to the screen.

Positioning The File Pointer :

Fseek is used to position the file pointer at the appropriate record in the file before reading or writing a record. The function fseek sets the file position for a stream. Subsequent reads and writes will access data in the file beginning at the new position

SYNTAX: int fseek (FILE *fp, long offset,int origin)

- The position is set to the offset from the origin
- fseek returns non zero on error

fseek takes 3 arguments

- fp the file pointer for the file in question
- offset is an offset in a file we want to seek to
- origin is the position that the file pointer is set to origin can be either
    - SEEK_SET - beginning of file
    - SEEK_CUR - current position in file
    - SEEK_END - end of file

It may be necessary when performing a number of different types of access to a single file to reset the file pointer in certain instances.

Ex: searching for a particular record in a file may leave the file pointer in the middle of the file.

To write a new record to this file the pointer should be reset to point to the end of file.

rewind :

The function rewind resets the file pointer back to the start of file.

Syntax : void rewind (FILE *fp)
rewind(fp) is equivalent to fseek(fp,0,SEEK_SET).

feof :

The function feof tests for end of file.

Syntax : int feof(FILE *fptr)

Feof accepts a pointer to a FILE, It returns non-zero if EOF and zero otherwise.
It is used when reading a file to check whether the EOF has been reached

These are three main areas where text and binary mode files are different. There are

Handling of New Lines :

In Text Mode, a newline character is connected into the carriage return –linefeed combination before being written to the disk. Likewise, the carriage return-linefeed combination on the disk is converted back into a newline when the file is read by a C program.

In Binary Mode, these conversions will not take place.

End of File :

In Text Mode, a special character, whose ascii value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function would return the EOF signal to the program.

In Binary Mode, these is no such special character present in the binary mode files to mark the end of file. The Binary Mode files keep track of the end of file from the number of characters present in the directory entry of the file.

The file that has been written in text mode is read back only in text mode. Similarly, the file that has been written in binary mode must be read back only in binary mode.

Storage of Numbers :

In Text Mode, numbers are stored as strings of characters. Thus 1234 even though occupies two bytes in memory, when transferred to the disk would occupy four bytes, one byte per character.

In Binary Mode, the number will be stored in Binary format. Each number would occupy same number of bytes on disk as it occupies in memory.

STEPS IN WORKING WITH FILES:

1. Establish a Buffer Area
2. Opening a File
3. Reading from a File or Writing to a File
4. Closing a File.

I. Establish a Buffer Area :

The first step is to establish a buffer area, where the information is temporarily stored while being transferred between the computers memory and the file.
The Buffer Area is established by writing

```
FILE *ptrvar;
```

FILE is a Structure which has been defined in the header file "stdio.h" (Standard input/output header file), which is a link between the Operating System and Program. It is necessary to include the header file "stdio.h". The FILE Structure contain information about the file being used, such as its current size, its location in memory etc. This buffer area allows information to be read from or written to file more fastly.

*prtvar is a pointer variable, which contains address of the Structure FILE.

2. Opening a File :

A File must be opened before it can be created or processed. This association the file name with the buffer area. It also specifies how the file will be utilized i.e. read-only file, write-only file, or read/write file.

We use the library function fopen() for opening a file.

```
ptrvar = fopen(filename, mode)
```

where filename and mode are strings. Filename represent the name of the file, mode represents how the file opened i.e., read only, write only etc.

The fopen() function returns a pointer to the begins of the buffer area associated with the file. A NULL value is returned if the file cannot be created or when existing file cannot be found.

The different file opening modes area listed below.

Mode	Meaning
"r"	Open an existing file for reading only
"w"	Open a new file for writing only. If the file exists, its contents are over written.
"a"	Open an existing file for appending (i.e., for adding new information at the end of the file). A new file will be created if the file does not exists.
"r+"	Open an existing file for both reading and writing.
"w+"	Open a new file for both reading and writing. If the file exists, the contents are over written
"a+"	Open an existing file for both reading and appending. A new file will be created if the file does not exists

To open the file in Binary file mention the mode along with a extra letter 'b' as "rb", "wb", "rb+" and so on.

3. Reading /Writing form/to a File :

(i) After opening the file we can read the characters from a file or write the characters into the file by using the two library functions fgetc() and fputc().

fgetc() ⇒ reach a character from the current position and advances the pointer position to the next character, and returns the character that is read.

Eg:- ch = fgetc(ptrvar)

Fputc() ⇒ Writes a character into the file.

Eg:- fputc(ch,ptrvar)

(ii) fscanf() and fprintf() are two functions used for formatted reading and writing of characters, Strings, integers and floats.

(iii) fread() and fwrite():

These are two functions to read and write structures with file in binary mode.

The Syntax is as follows:

```
fwrite(&record, sizeof(record),no of records, fileptr);
fread(&record, sizeof(record),no of records, fileptr);
```

The first argument is the address and the structure.
The second argument is the size of the structure in bytes.
The third argument is the number of structures to read or write.
The last argument is pointer to the file.

4. Closing the File :

Finally, the file must be closed at the end of the program. This can be done by using the function fclose().

fclose(ptrvar);

It is good programming practice to close a file explicitly, even though C compiler will automatically close the file.

/* Program to Create a File */

```
main()
{
        FILE *fp;
        char ch;
        fp = fopen("chars",'w');
        printf("Enter the characters(enter * to stop)\n");
        while((ch=getchar())!='*')
```

```
                putc(ch,fp);
        printf("File Created");
        fclose(fp);
    }
```

/* Program to Create a read an Existing File */

```
    #include<stdio.h>
    main()
    {
        FILE *fp;
        char ch;
        fp = fopen("chars",'r');
        printf("The contents of the file are:\n");
        while((ch=fgetc(fp))!=EOF)
                printf("%c",ch);
        fclose(fp);
    }
```

/* Program to Copy One File to Another File */

```
    #include<stdio.h>
    main()
    {
        FILE fs,ft;
        char ch;
        fs = fopen("source",'r');
        if(fs = =NULL)
        {
                puts("Cannot open source file");
                exist();
        }
        fr = fopen("target",'w');
        if(ft = =NULL)
        {
                puts("Cannot open target file");
                fclose(fs);
                exist();
        }
        while((ch=fgetc(fs))!=EOF)
                putc(ch,ft);
        printf("File Copied");

        fclose(fs);
        fclose(ft);
        }
```

/* Program to illustrate the use of fprintf() */

```
        #include<stdio.h>
        main()
        {
                FILE *fp;
                char itemname[20];
                int qty;
                float rate;
                char choice = ‘y’;

                fp = fopen (“Item”,”w”);

                while (choice = = ‘y’)
                {
                        printf(“Enter Item Name, Quantity and Rate \n”);
                        scanf(“%s%d%f”,itemname,&qty,&rate);
                        fprintf(fp, “%s%d%f\n”,itemname, age,rate);
                        printf(“Do you want to Continue (y/n)”);
                        fflush(stdin);
                        choice = getch();
                }
                fclose(fp);
        }
```

/* Program to illustrate the use of fscanf() */

```
        #include<stdio.h>
        main()
        {
                FILE *fp;
                char itemname[20];
                int qty;
                float rate;

                fp = fopen (“Item”,”r”);

                while (fscanf(fp,“%s%d%f”,itemname,&qty,&rate)!=EOF)
                {
                        printf(“Item Name : %s\n”,itemname);
                        printf(“Quantity =%d\n”,qty);
                        printf(“Rate = %f\n”,rate);
                }
        fclose(fp);
    }
```

Till now, we have seen examples of writing individual items into the file. Now we see how to write a record into a file and read a record from the file by using fprintf() and fscanf() respectively.

Here is the program.

/* Program for writing records into a file using Structures */

```
    #include<stdio.h>
    main()
    {
        FILE *fp;
        char choice = ‘y’;
        struct emp
        {
            char name[40];
            int age;
            float bs;
        };
        struct emp e;
        fp = fopen(“emp.dat”, “w”);
        if(fp = = NULL)
        {
            puts(“Cannot open file”);
            exit();
        }
        while (choice = =’y’)
        {
            printf(“Enter Name, age and Basic Salary”);
            scanf(“%s%d%f”, e.name, &e.age, &e.bs);
            fprintf(fp, “%s%d%f\n”,e.name,e.age,e.bs);
            printf(“Do you want to add one more record (y/n)”);
            fflush(stdin);
            choice =getch();
        }
        fclose(fp);
    }
```

/* Program to read records from file using Structures */

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

```
main()
{
        FILE *fp;
        struct emp
        {
                char name[40];
                int age;
                float bs;
        };
        struct emp e;
        fp = fopen(“emp.dat”, “r”);
        if(fp = = NULL)
        {
                puts(“Cannot open file”);
                exit();
        }
        while (fscanf(fp,“%s%d%f”, e.name, &e.age, &e.bs)!=EOF)
        {
                printf(“\n Employee Name : %s\n”,e.name);
                printf(“\n Employee Age:%d”,e.age);
                printf(“\nBasic Salary = %f”,e.bs);
        }
        fclose(fp);
}
```

In the above programs the Structure elements are handled individually. Since the above program is opened in text mode it would occupy more number of bytes. If the number of fields in the Structure increases, writing Structures using fprintf(), or reading them using fscanf() becomes complex. So overcome these disadvantages we open the file in binary mode and also handle the structures collectively as a single one by using fread() and fwrite() function.

/* Writing records into file in Binary Mode */

```
#include<stdio.h>
main()
{
        FILE *fp;
        char choice = ‘y’;
        struct emp
        {
                char name[20];
                int age;
                float bs;
            };
            struct emp e;
            fp = fopen(“Employee.dat”, “wb”);
            if(fp = = NULL)
            {
                puts(“Cannot open file”);
                exit();
            }
            while (choice = =’y’)
            {
                printf(“Enter Name, age and Basic Salary”);
                scanf(“%s%d%f”, e.name, &e.age, &e.bs);
                fwrite(&e,sizeof(e),1,fp);
                printf(“Do you want to add one more record (y/n)”);
                fflush(stdin);
                choice =getch();
            }
            fclose(fp);
        }
```

/* Read the records from Binary File */

```
        #include<stdio.h>
        main()
        {
            FILE *fp;
            struct emp
            {
                char name[20];
                int age;
                float bs;
            };
            struct emp e;
            fp = fopen(“Employee.dat”, “rb”);
            if(fp = = NULL)
            {
                puts(“Cannot open file”);
                exit();
            }
            while (fread(&e,sizeof(e),1,fp)
            {
                printf(“\n Employee Name : %s\n”,e.name);
                printf("\n Employee Age:%d",e.age);
                printf("\nBasic Salary = %f",e.bs);
        }
        fclose(fp);
}
```

/* Program for copying one file to another file using command line arguments */

```
#include<stdio.h>
main(int argc, char *argv[])
{
        FILE *fs, *ft;
        Char ch;

        if (argc!=3)
        {
                puts("In sufficient arguments");
                exit();
        }
        fs = fopen(argv[1], "r");
        if (fs= =NULL)
        {
                puts("Cannot open source file");
                exit();
        }
        ft = fopen(argv[2],"w");
        if (ft = =NULL)
        {
                puts("Cannot open target file");
                fclose(fs);
                exit();
        }
        while ((ch=fgetc(fs))!=EOF)
                fputc(ch,ft);
        fclose(fs);
        fclose(ft);
}
```

If the above program is saved as filecopy.c and executed at command line as

```
        filecopy first.txt        second.txt
```

argv[1] will be replaced with first.txt
argv[2] will be replaced with second.txt

The advantages of the above program are

a) There is no need to recompile the program every time.

b) Source and target files can be passed from command prompt.
c) Once executable file is created we cannot after the original source file.

/* Program to add two numbers using command line arguments */

```
main(int argc, char *argv[])
{
        int a,b,c;
        a = atoi(argv[1);
        b= atoi(argv[2]);
        c = a+b;
        printf("The sum of two numbers is:%d",c);
}
```

If program is saved as add.c then to execute

```
add 20 30
```

Note:- Since 20 and 30 are entered from Command line, they are treated as strings, so convert them into numbers using atoi() function.

STANDARAD INPUT/OUTPUT :

In 'C' a file is basically stream of bytes(more commonly referred as stream),which can be interpreted by a 'c' program.When a 'c' program is started ,the operating system is responsible for opening three file(streams).

When program execution begins, three files and their associated streams are connected to the program automatically. They are

- the standard input stream
- the standard output stream
- the standard error stream

Standard input: It is connected to the keyboard & it enables a program to read data from the keyboard.

Standard output: It is connected to the screen & it enables a program to write data to the screen.

Standard error: It is connected to the screen & all error messages are output to standard error.

Standard input and standarad output functions are handled through several library functions such as scanf(),getchar(),gets(),printf(),etc.

ERROR HANDLING DURING I/O OPERATIONS:

It is possible that an error may occur during I/O operations on a file. Typical error situations include the following:

- Trying to read beyond the end-of-file mark
- Device Overflow
- Trying to use a file that has not been opened
- Trying to perform an operation on a file, When the file is opened for another type of operation.
- Opening a file with an invalid filename.
- Attempting to write to a write-protected file.

If we fail to check such a read and write errors a program may behave abnormally when an error occurs an unchecked error may result in a premature termination of the program or incorrect output.
We have two library functions FEOF and FERROR that can help us to detect I/O errors in the files.
The FEOF function can be used to test for an End Of File condition. It takes a file pointer as it's only argument and returns a non zero integer value, if all of the data from the specified file has been read, and returns zero otherwise. If fp is pointer to file that has just been open for reading, then the statement

```
If (FEOF(fp))
Printf("End of data");
```

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

FERROR: The FERROR function reports the status of the file indicated. It also takes a file pointer as it's argument and returns a non zero integer if an error has been detucted up to that point during process it returns zero otherwise.

Example:

```
If(FERROR(fp)!=0)
Printf(" an error has occurred");
```

The above statement print the error message if the reading is not successful.

If a file is open using FOPEN function, A file pointer is returned. If the file cannot be opened for some reason then the function returns a null pointer. This facility can be used to test whether a file has been opened or not.

Example:

```
If(fp==NULL)
Printf("File could not be opened");
```

FORMATTED INPUT/OUTPUT FUNCTIONS :

There are two formatting input/output functions that is scanf and printf.The scanf function receives a text stream from the keyborad and converts in to data values to be stored in variables.The printf function receives data values from the program and converts them into text stream to be displayed on the monitor.

These two functions can be used only with the keyboard and monitor.The 'C' library defines two more general functions that is fscanf and fprintf that can be used with any text stream.

Formatting Functions

Terminal input/output

```
Scanf("control string",…);
Printf("control string",…);
```

General input/output

```
Fscanf(stream_pointer, "control string",…);
Fprintf(stream_pointer, "control string",…);
```

Stream pointer:

Stream pointer is the pointer to the streams that has been declared and associated with atext file.

Ex:

```
FILE * spIn;
…
spIn=fopen("file name","r");
…
```

```
fscanf(spIn,”format string”,address list);
```

While we normally read data from the terminal keyboard using scanf,we can also use fscanf.When we use fscanf we must specify that the stream pointer is stdin,for example

```
fscanf(spIn,”format string”,address list);
```

Ex:

```
FILE * spOut;
…
spOut=fopen(“file name”,”w”);
…
fprintf(spOut,”format string”,address list);
```

We can use fprintf to print to the terminal monitor by specifying the stream stdout, as shown below

```
fprintf(spOut,”format string”,address list);
```

Format Contol Strings:

Input and output functions for text files use a format string to dexcribe how data are to be formatted when read or written.The format control string consist of three types of data ,which may be repeated: whitespace ,text characters and conversion specification. That describes how the data are to be formatted as they are read or written.

Whitespace :

A whitespace character in an input format string causes leading whitespace characters in the input to be discarded.A whitespace character in an output format string is copied to the output stream.

Text :

Any text character other than a white space in an input format string must match exactly the next character of the input stream. If it does not match, a conflict occurs that causes the opration to be terminated. The conflicting input character remains in the input stream to be read by the next input operation on that stream.

Text characters in an output format string are copied to the output stream. They are usually used to displat messages to the user or to label data being output.

Conversion specification:

The conversion specification consists of a percentage character(%), optional formatting instructions and a conversion code. With one exception each conversion specificartion must have a matching parameter in the parameter list that follows the format string. The type in the conversion specification and the type of the parameter must match.

Conversion specifications can have up to six elements ,they are

- Conversion specification token(%)
- Flag
- Minimum width
- Precision
- Size
- Conversion code

First and last element elements are compulsory and the remaining elements are optional.

Conversion codes:

The conversion code specifies the type of data that are being formatted for input it specifies the type of variable in to which the formatted data are stored. For output specifies the type of data in the parameter associated with the specification.

INPUT FORMATTING:

The scanf and fscanf functions read text data and convert the data to the types specified by a format string. Only difference between thejm is that scanf reads data from the standard input unit(keyboard) and fscanf reads the input from file specified by the first parameter. This file can be standard input (stdin).

The name scanf stands for scan formatted, the name fscanf stands for file scan formatted. The functions have the following formats:
Scanf (“format string”,address list).
fscanf (sp,”format string”,address list).

Where sp is the address of a stream defined as type FILE*, format string is a string containing formatting instructions and the address list specifies the addresses where the data are to be stored after they have been formatted. A comma must separate the format string from the variable list.

INPUT DATA FORMATTING:

The conversion operation processes input characters until any of the following occurs

- End Of File is reached.
- An inappropriate character is encountered
- The number of characters read is equal to an explicitly specified maximum field width.

INPUT CONVERSION SPECIFICATION:

FLAG:

There is only one flag for input formatting,The assignment suppression flag(*). The assignment suppression flag tells scanf that the next input field is to be read but not stored. It is discarded.

Example:

```
Scanf("%d%*c%f', &x,&y);
```

WIDTH:

The width specifies the maximum width of the input(in characters).This allows us to break out a code that may be stored in the input without spaces.

SIZE:

A size specification is a modifier for the conversion code. Used in combination with the conversion code,specifies the type of the associative variable.

CONVERSION CODES:

Integer(d): The decimal (d) format code accepts a value from the input stream and formats it in to the specified variables. It reads only decimal digits and an optional plus or minus sign is the first character of the value.

Integer(i): The integer format code (i) allows the user to create decimal ,octal or hexadecimal numbers. Numbers starting with any digit other than zero are read and stored as decimal values. Numbers starting with zero are interpretes as octal values and are converted to decimal and stored. Hexadecimal numbers must be prefixed with 0x (or) 0X, The hexadecimal value is converted to decimal and stored.

Octal and hexadecimal(o,x): The octal (o) and hexadecimal (x) conversion codes perform unsigned conversion. For octal only valid input digits are 0…7. For Hexadecimal input valid digits are 0…9, a…f and A…F.

Scientific notation (e,g,a): The C languages uses three real format codes for scientific notation. In scientific notation the significand and exponent are specify separately. The significant part is a floating point number that contains as many significant digits as possible.

Count(n): To verify the number of input characters we use the n-conversion code. The code requires a matching variable address in to which scanf places the count of the characters input.

The Common Input Format Mistakes: Two common mistakes while formatting input stream, They are

Invalid Address
Datatype conflict

Invalid address: The common mistake is to use a data value rather than an address for the input parameter

Data type conflict: It is conflict between the format stream and the input stream. It occurs, For example when the format string calls for a numeric value and the input stream contains alphabetic character then scanf operation aborts.

OUTPUT FORMATTING (PRINTF AND FPRINTF) :

Two print formatted functions display output in human readable form under the control of a format string that is very similar to the format string in scanf.

Synatax: Printf("format string ", value list);
Fprintf(sp,"formatstring", value list);

One of the first difference is thatg the value list is optional .Where as you always need a variable when you are reading,in many situations ,such as user prompts ,displays strings without a value list.

Ex:
```
printf("\n welcome to calculator\n");
Printf("\nThe answer is %6.2f\n",x);
Printf("thank you for using calculator");
```
The fprintf function works just like printf except that it specifies the file in which the data will be displayed.

Print Conversion Specifications:

Printf family contains printf conversion specifications they are:
Flags
Sizes
Precision

Flags:

- **-** Value is left justified (default is right justified). Overrides the 0 flag.
- + Forces the sign (+ or -) to always be shown. Default is to just show the - sign. Overrides the space flag.

Width:
The width of the field is specified here with a decimal value. If the value is not large enough to fill the width, then the rest of the field is padded with spaces (unless the 0 flag is specified). If the value overflows the width of the field, then the field is expanded to fit the value. If a * is used in place of the width specifer, then the next argument (which must be an **int** type) specifies the width of the field. Note: when using the * with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted.

Precision:
The precision begins with a dot (.) to distinguish itself from the width specifier. The precision can be given as a decimal value or as an asterisk (*). If a * is used, then the next argument (which is an **int** type) specifies the precision. Note: when using the * with the width and/or precision specifier, the width argument comes first, then the precision argument, then the value to be converted. Precision does not affect the c type.

/*Create a file containing customer records */

```
#include<stdio.h>
#include<string.h>
typedef struct {
        int mm;
        int dd;
        int yy;
}date;

typedef struct{
        char name[80];
        char street[80];
        char city[80];
        int acc_no;
        char acc_type;          /* C =current, O =overdue D =delinquent */

        float old_bal;
        float new_bal;
        float payment;
        date lastpayment;
}record;
record read(record customer);/*function prototype */

FILE *fp;

main()
{

        record cust;
        char choice;

        fp = fopen("custold","w");

        printf("CUSTOMER BILLING SYSTEM \n\n");
        printf("Enter today date (mm/dd/yyyy)");

        scanf("%d%d%d",&cust.lastpayment.mm,&cust.lastpayment.dd,
        &cust.lastpayment.yy);
        cust.new_bal=0;
        cust.payment =0;
        cust.acc_type='C';

        do
        {
                printf("Enter Customer Name:");
                scanf("%s",cust.name);
                printf("Enter Street:");
                scanf("%s",cust.street);
                printf("Enter City:");
                scanf("%s",cust.city);
                printf("Enter Account Number:");
                scanf("%d",&cust.acc_no);
                printf("Enter Current Balance:");
                scanf("%f",&cust.old_bal);

                fwrite (&cust, sizeof(cust),1,fp);

                printf("Do you want to continue (y/n)");
                fflush(stdin);
                scanf("%c",&choice);
        }while(choice = ='y');
        fclose(fp);
}
```

/* Updating the file containing customer records */

```
#include<stdio.h>
#include<string.h>
typedef struct {
        int mm;
        int dd;
        int yy;
}date;

typedef struct{
        char name[80];
        char street[80];
        char city[80];
        int acc_no;
        char acc_type;          /* C =current, O =overdue D =delinquent */

        float old_bal;
        float new_bal;
        float payment;
        date lastpayment;
}record;
record update(record cust);

FILE *fp_old, *fp_new;
int month, day, year;           /*Declaring Temporary Variables*/

main()
{
        record cust;

        fp_old = fopen("custold","r");
        fp_new=fopen("custnew","w");
                clrscr():
                printf("CUSTOMER BILLING SYSTEM \n\n"):
                printf("Enter today date (mm/dd/yyyy)");
                scanf("%d%d%d", &month,&day,&year);

                while(fread(&cust,sizeof(cust),1,fp_old))
                {
                        cust = update(cust);    /*Calling Update Function*/
                        fwrite(&cust,sizeof(record),1,fp_new);
                        fread(&cust,sizeof(record),1,fp_old);
                }
        fclose(fp_old);
        fclose(fp_new);
}
record update (record cust)
{
        printf("\n\n Name: %s",cust.name);
        printf("Account Number: %d\n",cust.acc_no);
        printf("Enter current payment");
        scanf("%f",&cust.payment);

        if(cust.payment>0)
        {
                cust.lastpayment.mm=month;
                cust.lastpayment.dd = day;
                cust.lastpayment.yy = year;
                cust.acc_type = (cust.payment<0.1 *cust.old_bal) ? 'O' : 'C';
        }
        else
                cust.acc_type = (cust.old_bal>0) ? 'D' : 'C';

        cust.new_bal = cust.old_bal – cust.payment;
        printf("New balance: %7.2f",cust.new_bal);
        printf("   Account  Status  : ");
        switch(cust.acc_type)
        {
                case 'C'        : printf("Current \n");
                                  break;

                case 'O'        : printf("Overdue \n");
                                  break;

                case 'D'        : printf("Delinquent \n");
                                  break;

                default         : printf("ERROR    \n");
        }
        return(cust);
}
```

/* Menu driven program for adding, modifying, deleting and viewing records with a file */

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

```
struct emp
{
        int empno;
        char ename[20];
        float sal;
}e;

FILE *fp, *ft;
Long recsize;

main()
{
        char choice;
        int opt;

        fp =fopen(“employee.dat”,”rb+”);
        if(fp = =NULL)
        {
                fp = fopen(“employee.dat”,”wb+”);
                if(fp = =NULL)
                {
                        printf(“Cannot Open File\n”);
                        exit();
                }
                recsize = sizeof(e);
        }
        do
        {
                clrscr();
                printf(“1.Adding Records \n”);
                printf(“2.Modifying Records \n”);
                printf(“3.Deleting Records \n”);
                printf(“4.Viewing Records \n”);
                printf(“5.Exit \n”);
                printf(“Enter yout option(1/2/3/4/5) \n”);
                scanf(“%d”,&opt);

                switch(opt)
                {
                        case    1       :       addrecords();
                                                break;
                        case    2       :       modifyrecords();
                                                break;
                        case    3       :       deleterecords();
```

```
                                        break;
                case    4       :       viewrecords();
                                        break;
                case    5       :       fclose(fp);
                                        exit();
            }
            printf(“Do you want to continue (y/n) \n”);
            fflush(stdin);
            scanf(“%c”,&choice);
        }while(choice = = ‘y’);
}

addrecords()
{
        char more = ‘y’;
        fseek (fp,0,SEEK_END);
        while(more = =’y’)
        {
            printf(“Enter empno,ename and salary”);
            scanf(“%d%s%f”, &e.empno,e.ename,&e.sal);

            fwrite(&e,recsize,1,fp);

            printf(“Do you want to add one more record(y/n)”);
            fflush(stdin);
            more =getch();
        }
}
modifyrecords()
{
        char more =’y’;
        int temp_empno,t;
        while (more = =’y’)
        {
            printf(“Enter the empno of employee to modify\n”);
            fflush(stdin);
            scanf(“%d”,temp_empno);
            rewind(fp);
            while(fread(&e,recsize,1,fp) = =1)
            {
                if(temp_empno = = e.empno)
                {
                    printf(“\nEnter new name and new  salary”);
                    scanf(“%s%d”,e.ename,e.sal);
                    fseek(fp,-recsize,SEEK_CUR);
                    fwrite(&e,recsize,1,fp);
                }
            }
            printf("Do you want to modify one more record(y/n)");
            fflush(stdin);
            more =getch();
        }
}
deleterecords()
{
        char more ='y';
        int temp_empno;
        while (more = ='y')
        {
            printf("Enter the empno of employee to delete\n");
            scanf("%d",temp_empno);
            ft = fopen("temp.dat","wb");

            rewind(fp);
            while(fread(&e,recsize(e),1,fp) )
            {
                if(e.empno!= temp_empno)
                    fwrite(&e,recsize(e),1,fp);
            }
            fclose(fp);
            fclose(ft);

            remove("employee.dat");
            rename(temp.dat","employee.dat");
            fp = fopen("employee.dat","rb+");


            printf("Do you want to modify one more record(y/n)");
            fflush(stdin);
            more =getch();
        }
}
viewrecords()
{
        rewind(fp);
        while(fread(&e,recsize(e),1,fp) )
        {
            printf("Employee Number     :       %d\n",e.empno);
```

```
                printf("Employee Name        :        %s\n",e.ename);
                printf("Employee Salary      :        %f\n",e.sal);
        }
}
```

IMPORTANT QUESTIONS UNIT - VI

1. Write a C program to read a text file and to count
   a) number of characters
   b) number of words and
   c) number of sentences and write in an output file.
2. (a) What are the file I/O functions in C. Give a brief note about the task performed by each function.

   (b) Write a program to read an input file and count the number of characters in the input file.
3. (a) Distinguish between text mode and binary mode operation of a file.

   (b) Write a program to open a preexisting file and add information at the end of file. Display the contents of the file before and after appending.
4. (a) What is the task performed by fseek() function. What is its syntax? Explain each parameter in it.

   (b) Write a C program to read the text file containing some paragraph. Use fseek() and read the text after skipping 'n' characters from beginning of the file.
5. Write a C program to read information about the student record containing student's name, student's age and student's total marks. Write the marks of each student in an output file.
6. (a) How does an append mode differ from a write mode.

   (b) Compare between printf and fprinf functions.

   (c) Write a program to copy up to 100 characters from a file to an output array.
7. Write a C program to replace every fifth character of the data file using fseek() command.
8. Write a program for indexed sequential file for the employee database for the following operations:
   a) add record
   b) delete record
   c) search record based on the department.

9. Write a program to read the following data to find the value of each item and display the contents of the file.

Item	Code	Price	Quatity
Pen	101	Rs.20	5
Pencil	103	Rs.3	100

10. What is the purpose of library function feof( )? How feof( ) is utilized with in a program that updates an unformatted data file. Explain.

11. Describe types of files with an example.

12. (a) Write a C program to reads last 'n' characters of the file using appropriate file function.

(b) Write a C program to read a text file and convert the file contents in capital (uppercase) and write the contents in an output file.

13. (a) Distiguish between the following functions

(i) printf and fprintf

(ii) feof and ferror

(b) Write a program to copy the contents of one file in to another.

UNIT - VII

SEARCHING:

Searching for items is one of the most critical activities in many applications. Examples of few search activities are spell checking, searching for a name in list of names, opening files etc. The efficiency of a search depends on three things.

1. The size and organization of the collection of data we are searching.
2. The search algorithm that is used.
3. The efficiency of the test used to determine if the search is successful.

We will describe two searching methods in this session.

a) Linear Search which starts searching from the beginning of the list and checks each element until a match is found.

b) The second technique binary search which can only be used on a sorted list. It successively divides the array into two parts, then discards the part that cannot contain the element we are searching for.

Searching involves following:

a) **Search List:** The search list is simply a common term for the collection of elements that we are storing in. The elements in the list may or may not be ordered (ascending or descending).
b) **Target Element:** The element that we are searching for. (sometimes referred to as key element).

LINEAR SEARCH (SEQUENTIAL SEARCH):

Linear Search is the traditional method of searching. It is very simple but very poor in performance at times.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends. Otherwise, we will move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list.

For example consider the following list of elements.

5 9 7 8 11 2 6 4

Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 5. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons.

Function for linear search:

```
int LinearSearch(int a[], int n, int key)
{
        int i = 0;
        while(i<n-1)
        {
                if (a[i] == key)
                     return(i);
                i++;
        }
        return (-1);
}
```

The function returns the position of the target element, if it exists in the search list otherwise it returns a negative value (-1) to indicate unsuccessful search. The test i < n-1 provides a check to see that the array list is not crossed. For large lists, the performance of linear search degrades.

Table below lists performance for search list of N items.

Case	Meaning	Number of Iteration	Time Complexity
Best	The item to be searched is the first item in the list	1	θ(1)
Average	The item to be searched is found some where close to middle of list	N/2	θ(N)
Worst	The item to be searched is the last item in list, or it does not exist at all	N	θ(N)

Note:- Linear Search requires no ordering of elements in the list.

BINARY SEARCH:

Binary Search is a vast improvement over the linear search but it cannot always be used. For binary search to work, the items in the list must be in a sorted order (either increasing (ascending) or decreasing (descending) order similar to the way words in a dictionary are arranged.

The pre-requisite for Binary Search method is that the input elements list must be in the sorted order. The method starts with looking at the middle of the list. If it matches with the target element, then the search is complete Otherwise, the target element may be in the upper half or lower half. The search progresses with the upper half if the target element is greater than the middle element or with the lower half if the target element is less than the middle. The process is continued until the target element is found or the portion of the sub list to be searched is empty.

For example consider the ordered list of elements 2,5,8,9,11,15,17,23,43,65 to search the element 17(key).

Initially low = 1, high = 10. The middle element position is calculated by dividing the sum of lower bound and upper bound by 2 taking integer division. The search process can be seen as below,

```
2     5     8     9     11    15    17    23    43    65
↑                       ↑                             ↑
low                     mid                           high


2     5     8     9     11    15    17    23    43    65
                              ↑           ↑           ↑
                            low           mid         high


2     5     8     9     11    15    17    23    43    65
                              ↑ ↑   ↑
                            low |   high
                               mid


2     5     8     9     11    15    17    23    43    65
                                    ↑
                                    high
                                    low
                                          mid
```

Therefore the key element 17 is at 7th position.

The function for binary search can be written as

```
int  BinarySearch(int a[], int n, int key)
{
        int low, high, midpos;
        low = 0; high = n-1;
        while(low <= high)
        {
                midpos = (low+high)/2;
                if (key = = a[midpos])
                        return(midpos);
                else
                        if(key<a[midpos])
                                high = midpos –1;
                        else
                                low = midpos –1;
        }
        return(-1);
}
```

The function returns the position at which the item is found, otherwise –1 if the item is not found.

RECURSIVE BINARY SEARCH:

A recursive binary search differs from the iterative version mainly in that it requires the upper bound and lower bound indices to be passed as arguments.

Recursive Binary Search Function:

```
r_binarysearch( int a[], int key, int low, int high)
{
    int mid;
        if (low>high)
            return (-1)
                mid = (low +high)/2;
                if ( key = = a[mid])
                        return(mid);
                if (key < a[mid])
                        r_binarysearch( a, key,low, mid-1);
                else
                        r_binarysearch( a, key, mid+1,high);
}
```

The recursive function determines whether the search key lies in lower or upper half of the array, then it calls itself on appropriate half.

Each step of algorithm divides the list into two parts and the search continues in one of them and other is discarded. The search requires at most k steps where $2^k \geq n$ which results in time complexity of $k = \log_2 n$ i.e., O(log n) for average and worst cases.

SORTING TECHNIQUES:

Sorting:

The process of getting data elements into order is called Sorting.

We have different techniques in Sorting

1) Exchange Sort
2) Selection Sort
3) Insertion Sort
4) Quick Sort
5) Merge Sort
6) Tree Sort

1) EXCHANGE SORT:

Fundamental Mechanism in an exchange sort is to repeatedly make comparisons, and if required, to change adjacent items. Bubble Sort is an example of the exchange sort technique.

BUBBLE SORT PROCESS:

In this technique the first element in the list in tested with all the rest elements, and whenever the first element is found smaller than the rest element (to sort in descending order) then the swapping (exchanging) is done. This process will be repeated until the elements in list gets in sorted order

Bubble Sort for Strings:

```
void bubble_sort(char *str; int length)
{
        for(i=0; i<length; i++)
                for(j=i+1; j<length;j++)
                {
                    if (str[i]>str[j])
                    {
                         temp = str[i];
                         str[i] = str[j];
                         str[j] = temp;
                    }
               }
          printf("Sorted String is %s\n",str);
     }
```

Bubble Sort for Integers:

(For Ascending Order)

```
     bubblesort(int a[], int n)
     {
          int pass, j, noexchange;
          for(pass=0;pass<n-1;pass++)
          {
               noexchange=1;
               for(j=0;j<n-pass-1;j++)
               {
                    if(a[j]>a[j+1])
                    {
                         swap(a[j]>a[j+1]);
                         noexchange = 0;
                    }
               }
               if (noexchange);
                 return;
          }
     }
```

SELECTION SORT:

Suppose that items in a list are to be sorted in their correct sequence. Using Selection Sort, the first item is compared with the remaining n-1 items, and whichever of all is lowest, is put in the first position. Then the second item from the list is taken and compared with the remaining (n-2) items, if an item with a value less than that of the second item is found on the (n-2) items, it is swapped (Interchanged) with the second item of the list and so on.

Algorithm for Selection Sort:

```
Procedure Selection_Sort(A,N)
        For Index = 1 to N-1 do
        begin
                MinPostion ← Index
        For j = Index+1 to N do
        begin
        if  A[j] < A[MinPosition] then
                MinPosition ← j
        end
        swap (A[Index], A[MinPosition])
        end
        Return
```

Function to perform Selection Sort:

```
        void selection_sort(char *str, int len)
        {
                int i, j, k, swap;
                char temp;
                for(I=0;I<len;I++)
                {
                        swap = 0;
                        k=I;
                        temp = str[I];
                        for (j = I+1; j<len; j++)
                        {
                                if( str[j] < temp)
                                {
                                        k=j;
                                        temp = str[j];
                                        swap =1;
                                }
                        }
                        if (swap)
                        {
                                str[k] = str[i];
                                str[i] = temp;
                        }
                }
        }
```

Step by Step Iteration

I	Present Sequence
I = 0	b d a c
I = 1	a d b c
I = 2	a b d c
I = 3	a b c d

INSERTION SORT:

Insertion sort is implemented by inserting a particular element at the appropriate position. In this method, the first iteration starts with comparison of 1st element with the 0th element. In the second iteration 2nd element is compared with the 0th and 1st element. In general, in every iteration an element is compared with all elements before it. During comparison if it is found that the element in question can be inserted at a suitable position then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. This procedure is repeated for all the elements in the array.

Suppose the sequence is:

25, 17, 31, 13, 2

Step by Step Iteration

I	Present Sequence
I = 0	17 25 31 13 2
I = 1	17 25 31 13 2
I = 2	13 17 25 31 2
I = 3	2 13 17 25 31

Explanation:

a) In the first iteration the 1st element 17 is compared with the 0th element 25. Since 17 is smaller than 25, 17 is inserted at 0th place. The 0th element 25 is shifted one position to the right.
b) In the second iteration, the 2nd element 31 and 0th element 17 is compared. Since, 31 is greater than 17, nothing is done. Then the 2nd element 31 is compared with the 1st element 25.Again no action is taken as 25 is less than 31.
c) In the third iteration, the 3rd element 13 is compared with the 0th element 17.Since, 13 is smaller than 17, 13 is inserted at the 0th place in the array and all the elements from 0th till 2nd position are shifted to right by one position.
d) In the fourth iteration the 4th element 2 is compared with the 0th element 13. Since, 2 is smaller than 13, the 4th element is inserted at the 0th place in the array and all the elements from 0th till 3rd are shifted right by one position. As a result, the array now becomes a sorted array.

The complexity of the insertion sort algorithm is shown below:

Algorithm	Worst case	Average case	Best case
Insertion sort	$O(n^2)$	$O(n^2)$	n-1

QUICK SORT:

It is a fast sorting algorithm invented by C.A.R. Hoare. It approaches sorting in a radically different way from the other sorting techniques. It attempts to partition the data set into two sections, then sort each section separately adopting divide–and–conquer strategy.

Given a list of n elements, which is to be sorted, quick sort identifies an item (say k) in the list. It moves all the items in the list that have values less than k, to the left of k, and all the items that have a value greater than k to the right of k. Thus, we have two sub-lists, say list-1 and list-2 with in the main list, with k standing between list-1 and list-2. We then identify new k’s in the list-1 as well as list-2. Thus list-1 and list-2 each will be portioned further into two sub-lists, and on. This continues until the entire list is exhausted.

Algorithm for Quick Sort:

/* A is an array of N elements to be sorted. LowBound and UpBound are lower and upper bound indices of array. */

```
if(LowBound < UpBound) then
begin
i ← LowBound
j ← UpBound
pivot ← LowBound
while (i<j) do
begin
while((A[i]<=A[pivot]) and (i<UpBound)) do
        i← i+1
while(A[j]>A[pivot])do
        j←j-1
if(i<j) then
        swap(A[i],A[j])
end
swap(A[pivot],A[j])
Call QuickSort(A,LowBound,*j-1)
Call QuickSort(A,j+1,UpBound)
end
Return
```

Quick Sort Function:

```
void qsort(int a[], int lb, int ub)
{
        int i,j,pivot;
        if(lb<ub)
        {
                i = lb;
                j = ub;
                pivot = lb;
                while(i<j)
                {
                        while((a[i]<=a[pivot] && (i<ub))
                        i++;
                        while (a[j]>a[pivot])
                        j--;
                        if(i<j)
                                swap(&a[i],&a[j]);
                }
                swap(a[pivot],&a[j]);
                qsort(a,lb,a[j-1]);
                qsort(a,j+1,ub);
        }
}

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

MERGE SORT:

Merging means combining two sorted lists into one sorted list. For this the elements from both the sorted lists are compared. The smaller of both the elements is then stored in the third array. The sorting is complete when all the elements from both the lists are placed in the third list.

Two Lists before sorting:

```
          a                                   b

   11   2   9   13   57              25   17   1   90   3
```

The two sorted lists are:

```
          a                                   b

   2    9   11   13   57             1    3   17   25   90
```

Step by step iteration:

I	**Present Sequence (The third List)**
I=0	1
I=1	1 2
I=2	1 2 3
I=3	1 2 3 9
I=4	1 2 3 9 11

```
I=5     1  2  3  9  11  13

I=6     1  2  3  9  11  13  17

I=7     1  2  3  9  11  13  17  25

I=8     1  2  3  9  11  13  17  25  57

I=9     1  2  3  9  11  13  17  25  57  90
```

Suppose arrays a and b contain 5 elements each. Then merge sort algorithm works as follows:

a) The arrays a and b are sorted using any algorithm.
b) The 0th element from the first array, 2, is compared with the 0th element of second array 1. Since 1 is smaller than 2, 1 is placed in the third array.
c) Now the 0th element from the first array, 2, is compared with 1st element from the second array, 3. Since 2 is smaller than 3, 2 is placed in the third array.
d) Now the 1st element from the first array, 9, is compared with the 1st element from the second array, 3. Since 3 is smaller than 9, 3 is placed in the third array.
e) Now the 1st element from the first array, 9, is compared with the 2nd element from the second array, 17. Since 9 is smaller than 17, 9 is placed in the third array.
f) The same procedure is repeated till end of the array is reached. Now, the remaining elements from the other array are placed directly into the third list as they are already in sorted order.

The complexity of the merge sort algorithm is shown below:

Algorithm	Worst case	Average case	Best case
Merge sort	O(n log n)	O(n log n)	O(n log n)

The following program implements the merge sort algorithm:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5]={ 11, 2, 9, 13, 57 };
    int b[5]={ 25, 17, 1, 90, 3 };
    int c[10];
int i,j,k,temp;

clrscr();

printf("Merge sort.\n");

printf("\nFirst array:\n");
for(i=0;i<=4;i++)
   printf("%d\t",a[i]);

printf("\nSecond array:\n");
for(i=0;i<=4;i++)
   printf("%d\t",b[i]);

for(i=0;i<=3;i++)
{
   for(j=i+1;j<=4;j++)
  {
      if(a[i]>a[j])
     {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
     }
     if(b[i]>b[j])
     {
        temp=b[i];
        b[i]=b[j];
        b[j]=temp;
     }
  }
}

for(i=j=k=0;i<=9;)
{
      if(a[j]<=b[k])
           c[i++)=a[j++];
      else
          c[i++]=b[k++];

      if(j==5 || k==5)
          break;
}
```

```
for(;j<=4;)
   c[i++]=a[j++];

for(;k<=4;)
   c[i++]=b[k++];

printf("\nArray after sorting:\n");
for(i=0;i<=9;i++)
      printf("%d\t",c[i]);

getch();
}
```

HEAP SORT:

It is a sorting algorithm which was originally described by Floyed. It is a completely binary tree with the property that a parent is always greater than or equal to either of its children (if they exist).

Steps Followed:

a) Start with just one element. One element will always satisfy heap property.
b) Insert next elements and make this heap.
c) Repeat step b, until all elements are included in the heap.

Sorting:

a) Exchange the root and last element in the heap.
b) Make this heap again, but this time do not include the last node.
c) Repeat steps a and b until there is no element left.

Algorithms:

Building Heap:

```
Procedure BuildHeap(A,N)
For j = 2 to N do
begin
key ← A[j]
i ← trunc(j/2)
        while (i>0) and (A[i]<key) do
        begin
        A[j]←A[i]
        j←i
        i←trunc(j/2)
```

```
            end
            A[i]←key
            end
        return
```

Recreate the Heap:

Procedure ReHeap(A, k)

```
Parent←1
Child←2
Key←a[Parent]
Heap←False
while(Child<=k) and (not Heap) do
begin
    if(Child < k) then
        if A[Child + 1 ] > A[Child] then
            Child ← Child +1
        If A[Child]>Key then
            begin
            A[Parent]←A[Child]
            Parent←Child
            Child ←Parent*2
            end
        else
            Heap←True
        end
        A[Parent]←Key
        Return
```

Heap Sort Algorithm:

```
    Procedure HeapSort(A,N)
    Call Build Heap(A,N)
    For Pos = N to 2 in Steps of –1 do
        begin
        swap (A[1],A[Pos])
        Call ReHeap (A,Pos-1)
        end
        Return
```

IMPORTANT QUESTIONS UNIT - VII

1. (a) Write a C program to sort given integers using partition exchange sort.
   (b) Derive the time complexity of partition exchange sort.
2. (a) Write a C program to search for a given element in the integer array using Binary search.
   (b) Write a C program to sort the elements of an array using Tree sort method with suitable example.
3. (a) Write a C program to sort the elements of an array using Quick sort with suitable example.
   (b) What is the worst case and best case time complexity of the above program?
4. (a) Explain Quick sort with algorithm.

(b) Analyze the worst case performance of Quick sort and compare with selection sort.

5. What is a Heap? Show how would you perform a Heap sort using a Heap. Derive the time complexity of the Heap sort in the best and worst cases.

6. Write a C program that searches a value in a stored array using binary search. What is the time complexity of a binary search?

7. (a) Write a C program to merge two sorted arrays of integers.

(b) Derive the time complexity of merge sort?

8. What do you mean by sorting? Mention the different types of sorting, give some examples and explain any one in detail.

9. Compare the advantage and disadvantage of Bubble, Insertion and Selection sort.

10. Write a C program to explain selection sort. Which type of technique does it belong?

11. Write a program to sort the elements whose worst and average cases are O(n log n)?

12. Suppose that the list contains integers 1, 2, …. 8 in this order. Trace through the steps of binary search to determine what comparisons of keys are done in searching.

(i) To locate 3

(ii) To locate 4.5

13. Trace through the steps by hand to sort the following list in quick sort.

28 7 39 3 63 13 61 17 50 21

14. (a) Write and explain linear search procedure with a suitable example.

(b) Formulate recursive algorithm for binary search with its timing analysis.

15. Write in detail about the following

(i) Exchange sort

(ii) Binary search

(iii) Selection sort

(iv) Heap sort

UNIT - VIII

DATA STRUCTURES

Data structure is often confused with data types. Data type is a collection of finite set of permitted values and a set of operations on these values, where as, data structure is a study about the organizing the related data to enable us to work efficiently with data, exploring the relationships with in the data.

Data structure is a study of different methods of organizing the data and possible operations in these structures.

ABSTRACT DATA TYPES

In computing, an **abstract data type** or **abstract data structure** is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.

This approach doesn't always work on large programs in the real world, because these programs evolve as a result of new requirements or constraints. A modification to a program commonly requires a change in one or more of its data structures. For instance, a new field might be added to a personnel record to keep track of more information about each individual; an array might be replaced by a linked structure to improve the program's efficiency; or a bit field might be changed in the process of moving the program to another computer. You don't want such a change to require rewriting every procedure that uses the changed structure. Thus, it is useful to separate the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

Here are some examples.

- stack: operations are "push an item onto the stack", "pop an item from the stack", "ask if the stack is empty"; implementation may be as array or linked list or whatever.
- queue: operations are "add to the end of the queue", "delete from the beginning of the queue", "ask if the queue is empty"; implementation may be as array or linked list or heap.
- search structure: operations are "insert an item", "ask if an item is in the structure", and "delete an item"; implementation may be as array, linked list, tree, hash table, ...

There are two views of an abstract data type in a procedural language like C. One is the view that the rest of the program needs to see: the names of the routines for operations on the data structure, and of the instances of that data type. The other is the view of how the data type and its operations are implemented. C makes it relatively simple to hide the implementation view from the rest of the program.

STACKS & QUEUES

Stacks and Queues are very useful in numerous applications. For eg. Stacks are used in computers in parsing an expression by recursion; in memory management in operating system; etc. Queues find their use in CPU scheduling in printer spooling, in message queuing in computer network etc.

STACKS:

Stack is an ordered collection of data in which data is inserted and deleted at one end (same end). In stack the most recent elements are inserted first. For eg. plates are pushed on to the top and popped off the top. Such a processing strategy is referred to as Last-in-First-out or LIFO. A Stack is a data structure which operates on a LIFO basis, it is used to provide temporary storage space for values. Stack is most commonly used as a place to store local variables, parameters and return addresses. When a function is called the use of stack is critical where recursive functions, that is function that call themselves are defined.

Stack uses a single pointer 'top' to keep track of the information in the stack.

Basic operation is a stack are.

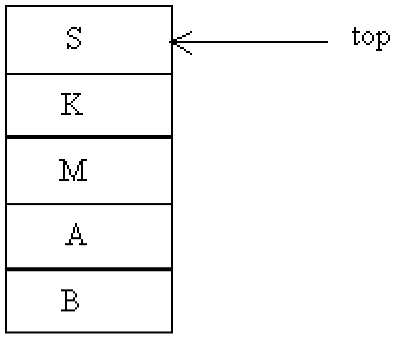
1. Insert (push) an item onto stack.
2. Remove (pop) an item from the stack.

A stack pointer keep track of the current position on the stack. When an element is placed on the stack, it is said to be pushed on the stack when an object is removed from the stack, it is said to be popped from the stack. Stack 'Overflow' occurs when one tries to push more information on a stack, and 'Underflow' occurs when we try to pop an item off a stack which is empty.

REPRESENTATION OF STACK:

Two standard methods of representing a stack are in terms of an array and a pointer linked list which is a away of representation using pointer.

The figure below illustrates an array implementation of a stack of 5 elements B, A, M, K, S.



□ □ □ □□□□□□□□

The Stack pointer is pointing to the top of the stack (called the top of the stalk). The size of the array is fixed at the time of its declaration itself. But, as the definition shows, stacks are dynamic data structures. They can grow and shrink during the operation. However, stack can be an implemented using arrays by specifying a maximum size. We require two things to represent stacks. First one is array to hold the item and second an integer variable to hold the index of the top element conveniently in C, we can declare stack as a structure to represent these two
□□□#define max 50

```
    struct stack
    {
        int top:
I       int items [max]:
    };
```

Now the stack s can be declared as struct stack S;

Note that the type of array elements in the above declaration is integer, but actually it can be of any other type. Since array subscripts starts from 0, the 50 stack elements in the above example reside in the location S.item[0] through S.items[49]. The top element can be accessed as S.item[top].

Basic Operations on a Stack:

Push:

Add an item to a stack moving the stack pointer to accommodate a new item. Push always places the element on the top of the stack
.

Pop:

Removes the most recently pushed item from the stack, moving the stack pointer top. Pop always removes top most element from stack.

Since a stack can contain only a finite number of elements, an error called a 'stack overflow' occurs when the stack is full and a push operation is performed. Similarly, an error called a stack underflow occurs when the stack is empty and a pop operation is performed.

These two conditions need to be checked before doing the respective operations.

Pushing Items on to the Stack:

Assume that the array element begins at 1 and the maximum elements of the stack are MAX. The stack pointer, say top, is considered to be pointing to the top elements of the stack. A push operation thus involves adjusting the stack pointer (top) to point to next tree slot and then copying data into that slot of the stack. Initially top is initialized to zero.

ALGORITHM: push an element onto a stack

Procedure push(s, MAX, Top, ITEM)
/* s is array with size MAX represents the stack. Top is a pointer to the top element of stack and ITEM is the item to be inserted into the stack*/

```
1.  if top>= MAX the then
       print 'stack overflow'
    return
2.  top ← top +1
3.  s[top] ← ITEM
4.  return
```

In the second step of algorithm the stack pointer is adjusted to point to the next free location in the stack, i.e. by incrementing the pointer top. The new element, ITEM, is inserted into the top of the stack in step 3.

Removing items from the Stack:

To remove an item, first extract the data from the top position in the stack and then decrement (subtract 1 from) the Stack pointer, top.

ALGORITHM: pop an element from stack

Function pop (s, top)

```
1.  if top =0 then
       print 'stack underflow'
    return.
2. K ← s[top]
```

3. top ← top–1
4. return (k)

INFIX, PREFIX, POSTFIX NOTATIONS FOR EXPRESSIONS:

In general, simple arithmetic expression can be represented in three ways : infix, prefix and postfix.

Consider the expression A+B.

Here A and B are operands and + is a binary operator. An operator is a binary operator if it is associated with two operands.

The 3 notations can be shown as

A+B infix
+AB prefix
AB+ postfix (or suffix or reverse polish)

In infix notation the operator symbol is between the two operands. In prefix, the operator precedes both operands. Here the prefixes pre- and post- indicate the relative position of the operator with respect to its own operands.

Consider the following two infix expressions:

A+(B*C) and (A+B)*C

We will see how to convert the expression into other two notations
(Brackets { } are used to indicate the partial conversion)

Infix to prefix:

Original expression	Intermediate expression	Expression after conversion
A +(B*C)	A + {*B C}	+A*BC
(A+B)*C	{+AB}*C	*+ ABC

Prefix to postfix:

Original expression	Intermediate expression	Expression after conversion
A + (B*C)	A + {BC*}	ABC * +
(A+B)*	{AB+}*C	AB + C*

We represent the exponential symbol as. ^. If parenthesis are not used then we can evaluate considering the operation–precedence rules.

The infix operator – precedence is as follows:

1. Parenthesis ()
2. Exponentiation ^ (Right to left)
3. Multiplication *, division / (left to Right)

4. Addition +, Subtraction – (left to Right)

Some example expressions in three notations.

Infix	Prefix	Post fix
(A+B)*C/D	* +AB/CD	AB+CD/*
A + (B*C)/D	+ /A *B C D	ABC*D/+
C /D*E	* /CDE	CD /E*
A+B*C +(D*E+F)*G	+ +A*BC*+*DEFG	ABC* + DE*F+G*+

Observe the parenthesis are never required to evaluate any expression represented on prefix or postfix notations. This is because the order in which the operations are to be performed is specified by the positions of operands and operators in the expression. So, the evaluation of postfix (or prefix) expression are very easy. In computers, usually the expressions written in infix notations are evaluated by first converting into its postfix expressions and then evaluate the postfix expression. Stacks are useful for the conversions and evaluations steps.

CONVERSION OF EXPRESSION FROM INFIX TO POSTFIX:

ALGORITHM: Conversion of Infix Expression to Postfix

1. Push to the end of X, where X is an arithmetic expression.
2. Check expression from left to right and follow the steps 3 to 6 for every element of X until the stack is vacant.
3. In case an operand is found, add it to R, where R is the equivalent postfix expression.
4. In case left '(' paranthesis is found push it onto the stack.
5. In case an operator is found, follow the following steps
    a) Continually pop from the stack and add each operator to R at the top of the stack. The pushed item should have the same or higher precedence than the former operator encountered.
    b) Append operator (recently encountered) onto the stack.
6. In case ')' right paranthesis is found, then follow the following steps.
    a) Continually pop from the stack and append every operator to R until left paranthesis is found.
    b) Eliminate the left paranthesis.

EVALUATION OF POSTFIX EXPRESSION:

Evaluation of postfix(i.e., Reverse polish) expression. An algebraic expression in postfix notation is evaluated as follows

- The expression is scanned from left to right one at a time
- If a number is encountered, it is pushed onto the stack

- If an operator is encountered, it is applied to the top two operands on the stack. After removing the operands used, the result is pushed on to the stacks.(Note that the operator is not pushed onto the stack)

*The above steps are repeated until the end of expression is reached. Now, the top element (of course, stack contains only one element at this stage) of a stack is popped which is the result of the expressions.

RECURSION:

Many examples of the use of recursion may be found. This technique is useful for the definition of mathematical functions and for the definition of data structures; one of the simplest examples of a recursive definition is that for the factorial function.

Factorial (n) = 1 if n=0
= n* Factorial (n-1) otherwise

QUEUES:

A queue is an ordered collection of elements, where the elements can be inserted at one end and deleted from another end. It works on the principle called First–In–First–Out (FIFO).

The information we retrieve from a queue comes in the same order that it was placed on the queue. The examples of queues are checkout line at supermarket cash register, line of cars waiting to proceed in some fixed direction at an intersection of streets. The queue can be implemented using arrays and linked lists.

OPERATIONS ON QUEUES:

Insertion: The operation of adding new items on the queue occurs only at one end of the queue called the **rear end.**

Deletion: The operation of removing items of the queue occurs at the other end called the **front end.**

The following is an algorithm for inserting an element into the queue.

The variable FRONT holds the index of the front item and REAR variable holds the index of the last item. Array elements begin at 1 and the maximum elements of the Queue is MAX. Q is the Queue. ITEM is the element to be inserted.

ALGORITHM: Inserting an element into Queue

```
Queue_insert(ITEM)
If REAR > = MAX then write 'Queue Overflow '
Return
REAR ← REAR +1
Q[REAR] ← ITEM
IF FRONT ==0 THEN
FRONT ← 1
Return
```

The following is the algorithm for detecting an element from the queue.

ALGORITHM: Deleting an elements from the Queue

```
Queue_delete( )
If FRONT =0 then
Write('Queue under flow').
Return
K ← Q[FRONT]
If FRONT = REAR then
Begin
        FRONT ←0
        REAR ←0
End
Else
        FRONT ← FRONT +1
Return K
```

CIRCULAR QUEUE:

When the items from a queue get deleted, the space for that item is never used. Those queue positions continue to be empty. This can be solved by circular queue.

In a circular queue, the items are arranged in the form of a circle. The circular queue does not have a beginning or end.

DE-QUEUE:

It is called as double ended queue. In dequeue the elements can be inserted and deleted from both the ends of a queue.

PRIORITY QUEUE:

This queue will not operate strictly on FCFS (first come first serve) bases, but on some complex priority scheme.

A priority queue is a queue that contains items that have some predefined ordering. Unlike the usual removal of the first item, when an item is removed from a priority queue, some priority is associated with that, this priority determines the order in which it exits the queue.

INTRODUCTION TO LISTS:

Arrays can be used to represent the simple data in which it is sequential. These representations had the property that successive nodes of the data object were stored a fixed distance apart.

If the element a_{ij} of a table was stored at location L_{ij} then a_{ij+1} was at the location L_{ij+c} for some constant c.

If the ith node in a queue was at a location L_i, then i+1th node was at location L_i+c mod n for circular representation.

If the top element of a stack was at location L_i, then the element beneath it was at L_i – c etc.

But when a sequential mapping is used for ordered lists, insertions and deletions of arbitrary elements becomes expensive.

To make these operations easier to handle, we use linked representations. Unlike a sequential representation where successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. That means, in a sequential representation the order of elements is the same as in the ordered lists, while in the linked representation these 2 sequences need not be the same. To access elements in the list in the correct order, with each element we store the addresses or location of the next element in that list. So in linked representation data item and pointer to the next element is grouped into a single field that is called node.

Node, is a collection of data, and links.

Difference between sequential and linked representation:

SEQUENTIAL REPRESENTATION`	LINKED REPRESENTATION
List Size is fixed	List Size is not fixed
Elements are stored in contiguous locations in memory.	Elements are not stored contiguously, but anywhere in the memory.
Each element contains only information.	Each element contains data and pointer to next element.
Insertions and deletions are difficult to perform.	Insertions and deletions are easy to perform.

Wastage of memory, if unused. No extra memory required for the elements.	No wastage of memory. Extra memory for pointer field.

SINGLE LINKED LIST:

Single Linked List is a collection of nodes. Each node contains 2 fields: I) info where the information is stored and ii) link which points to the next node in the list.

The node is like this:

Node

Info (or) data	Link (or) next

The operations that can be performed on single linked lists includes: insertion, deletion and traversing the list.

Inserting a node in a single linked list:

Insertion can be done in 3 ways:
1.Inserting an element in the beginning of the list.
2.Inserting in the middle and
3.Inserting at end.

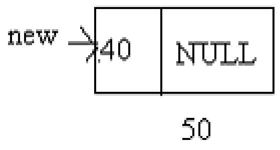
The algorithm for inserting an element in the beginning of a list:

Procedure insertbeg(x, first)
/* x – is an element first is the Pointer to the first element in a SLL */

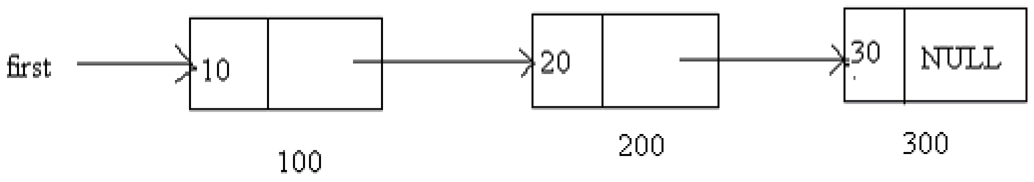
```
begin
    if avail = NULL        then
        Write(' Availability Stack underflow')
    return (first)
    else
        new ← avail;
        avail ← link(avail)
        info(new) ← x
        link(new) ← first
        return (new);
    end.
```

In the algorithm, first, the memory for a new node is available or not is checked out. Here avail is a pointer to the available memory.

If it is NULL then there is no memory. Otherwise a new node is created from available memory and it is stored in new. The new contains the address of the new node and avail moves forward to point to next available memory location. In the info field of new, x is stored and the link field of new points to the first element address of the list. Now, new becomes the pointer to the whole list.
This can be shown as:



(a)

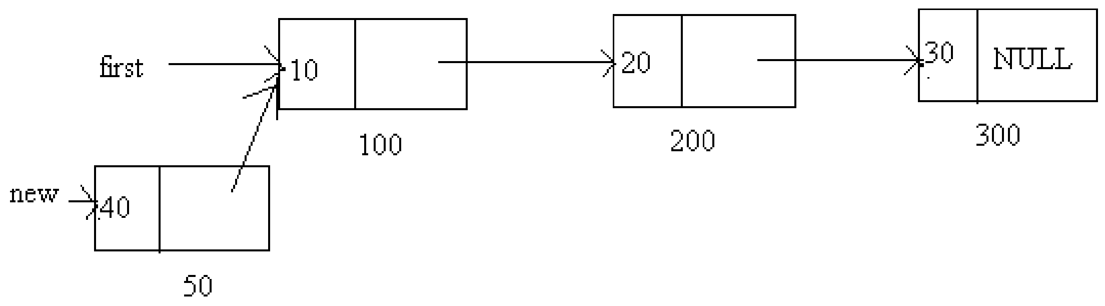


(b)

First is having value of 100, means it is Pointing the 100th memory location. The node at 100th location is having info as 10 and containing the address 200 of the next node. The second node at 200th location, contains 20 and address 300 of the next node. At,300th memory location, the node contains info as 30 and its link field is NULL. That means it is no more having any nodes. That is, the list is having 3 nodes with a starting address of 100.

Now, we created another node new that is having the address of 50 and its info is 40 and its link field is NULL.

After the call to insertbeg( ) function the list will look like.

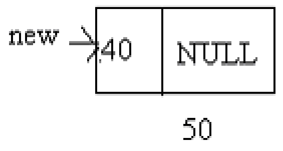


The insertbeg( ) function, inserts a node by storing the address of the first node of list in the link field of new and making the new address as the pointer to the list.

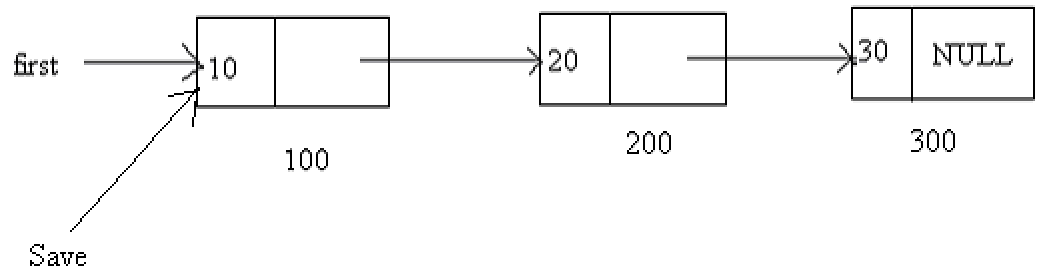
The algorithm for inserting an element at the end of the list:

```
Procedure insertend(x, first)
begin
        if avail = null then /* Checking for memory availability */
        write (' Availability Stack underflow');
                return(first);
        else    /* Obtaining the next free node */
        new ← avail;  /* Removing free node from available memory */
        avail ← link(avail)
        info(new) ← x/* Initializing the fields of new node */
        link(new) ← NULL
        if first = NULL  then  /* Is the list empty? */
        return(new);
        Save ← first /* Searching the last node */
        Repeat while link (Save) ≠ NULL
        Save ← link(Save)    /* Setting the link field of last node to new */
        Link (Save) ← new
        return (first)
end;
```

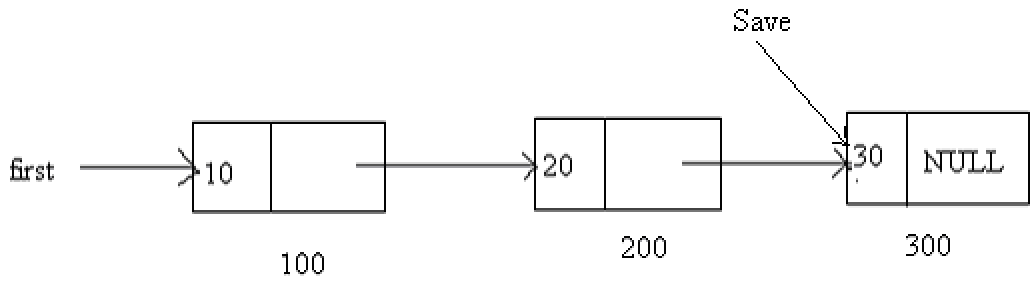
The Pictorial Representation:



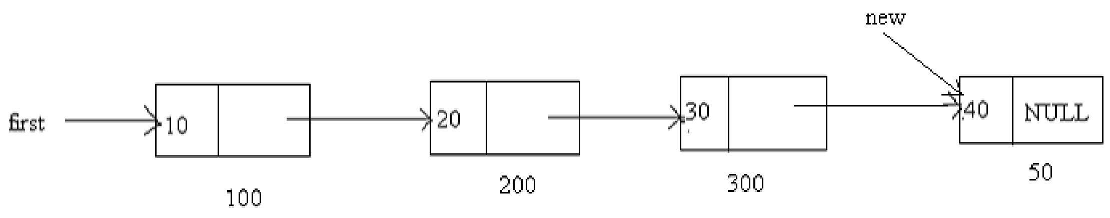
(a)



(b)



(c)



(d)

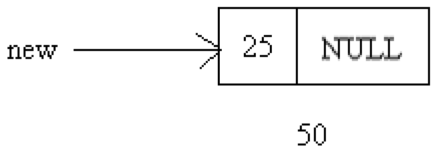
<u>first</u> (100) address is stored in <u>Save</u> to go through the end of the list, after reaching to the last node, set the address of <u>new</u> node in the <u>link</u> field of last node.

Inserting in the middle:

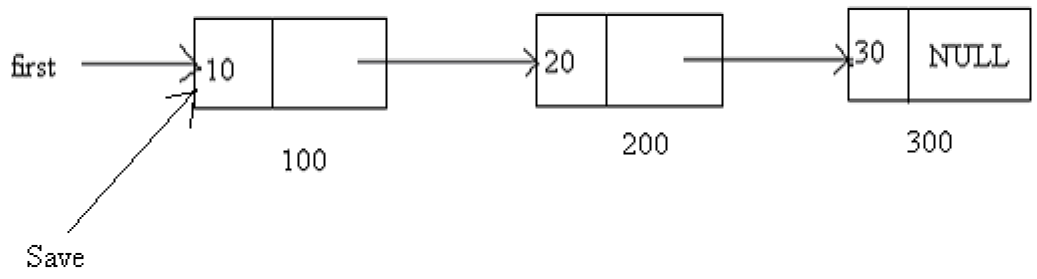
In this process, the address of first is stored in Save to go to the particular node at which the insertion is to be done. After reaching the particular node, set the link to point to new node, and set the link of new node to connect the remaining nodes.

ALGORITHM: The algorithm for inserting an element in the middle of a list:

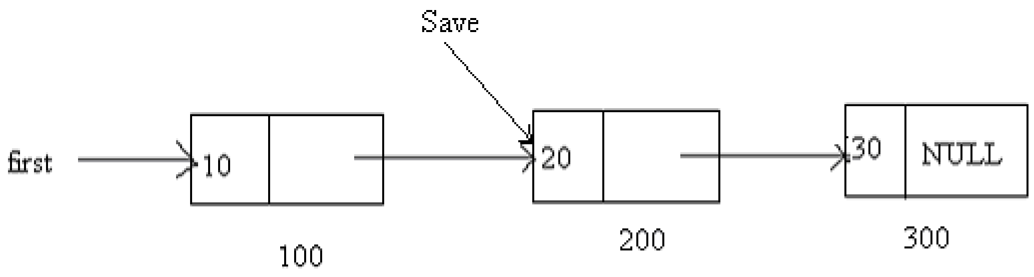
```
Procedure insertmid(x, first)
begin
    if avail = NULL  then
        write (‘ Availability Stack Underflow’);
        return (first)
    else    /* Obtain the address of next free node */
        new ← avail
        avail ← link(avail) /* Removing free node */
        info(new) ← x /* Copying Information into new node */
    if first = NULL then /* Checking whether the list is empty */
        link(new) ← NULL  /* list is empty*/
        return(new) /* if the new data precedes all other in the list */
    if info(new) ≤ info(first) then
        link(new) ← first
        return(new)    /* initialise temporary Pointer */
        Save ← first
        Repeat while link(Save) ≠ NULL and
            Info(link(Save)) ≤ info(new)
        Save ← link(Save)    /* Search for predecessor of new data */
            link(new) ← link(Save)    /* Setting the links of new and its
Predecessor*/
            link(Save) ← new
            return(first)
        end
```



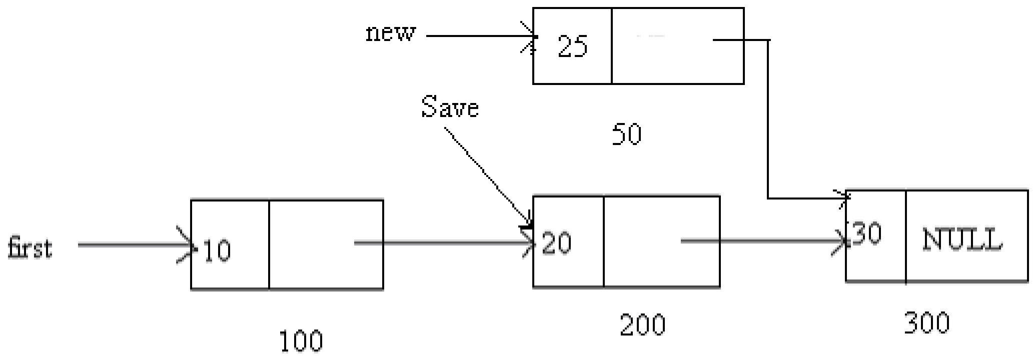
(a)


first
10
20
30
NULL
100
200
300
Save

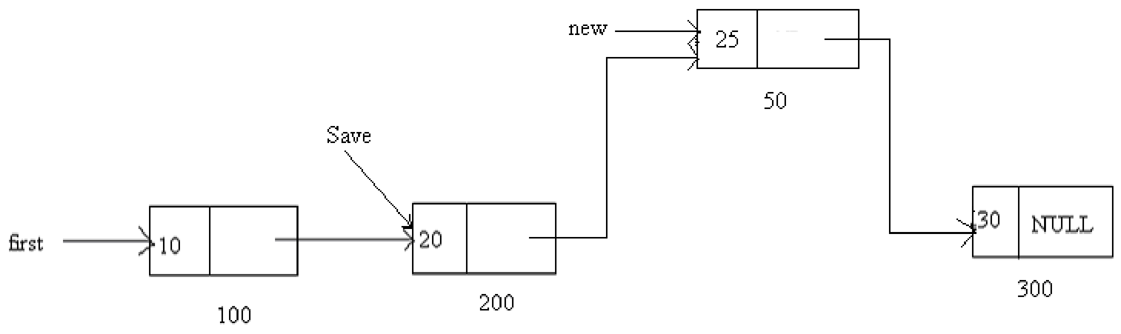
(b)


Save
first
10
20
30
NULL
100
200
300

(c)


new
25
50
Save
first
10
20
30
NULL
100
200
300

(d)



(e)

Deleting a node in a single linked list:

Deletion can be done in 3 ways:
1. Deleting an element in the beginning of the list.
2. Deleting in the middle and
3. Deleting at the end.

The algorithms for deleting an element in the list:

Procedure:

If the linked list is empty then write underflow and return
Repeat Step 3 while the end of the list has not been reached and the node has not been found.
Obtain the next node in the list and record its predecessor node.
If the end of the list has been reached then write node not found and return.
delete the node from the list.
Return the node to the availability area.

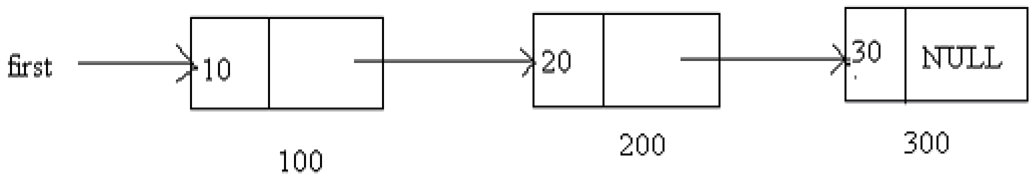
Algorithm:

Procedure deletelement(x, first)

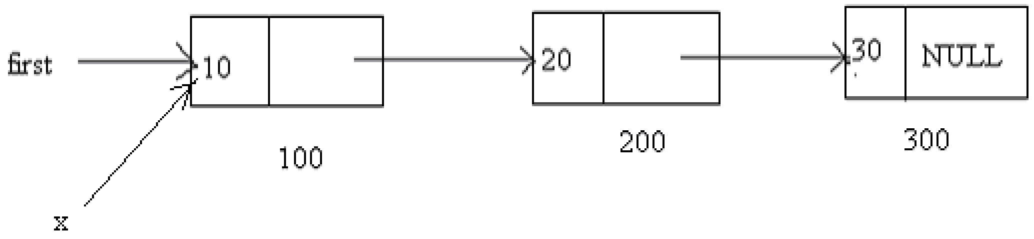
```
begin
        if first = NULL  then  /* Checking whether the list is empty */
                write(' Underflow');
                return
                temp ← first  /* Initializing search for x */
                while((temp ≠ x ) and
                (link(temp) ≠ NULL))
        begin
                pred ← temp
                temp ← link(temp)
        end
        if   temp ≠ x then /* if the node not found */
                write('node not found')
                return
        if (x = first) then / is x first node */
                first ← link(first)    /* delete the node at x */
        else    link(pred) ← link(x)
        link(x) ← avail   /* Return node to availability area */
        avail ← x
        return:
end.
```

In the algorithm, it first checks whether the list is empty, if it is empty it prints underflow message. Otherwise, it initializes a temporary pointer to first. Until the Predecessor of x node found, the temporary pointer is moved forward. If it reaches the end of the list, without finding the node of address x, it flashes an error message, stating 'node not found'. If x is the first address, then first node is deleted and now first points to second node. Otherwise it sets the links so that it deletes the node of address x.

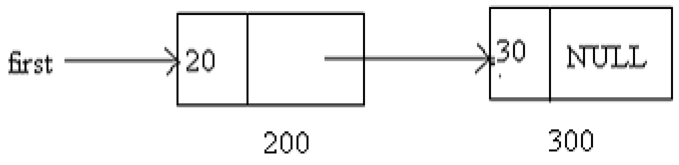
The original list is:



1. If x = 100 (First node)



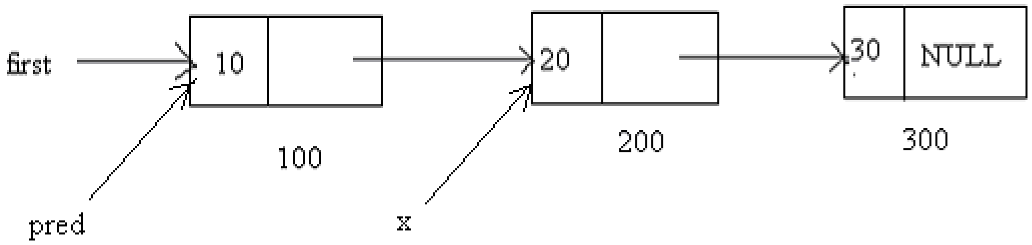
(a)



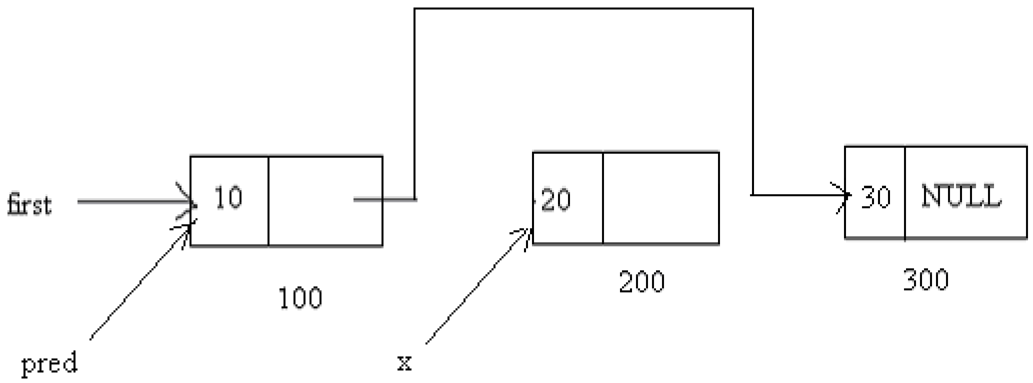
(b)

The node which is 100th memory address will be added to free space

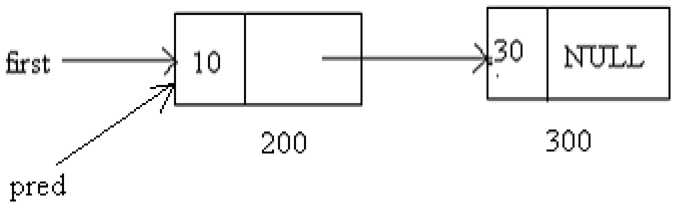
(ii) if x = 200



(a)

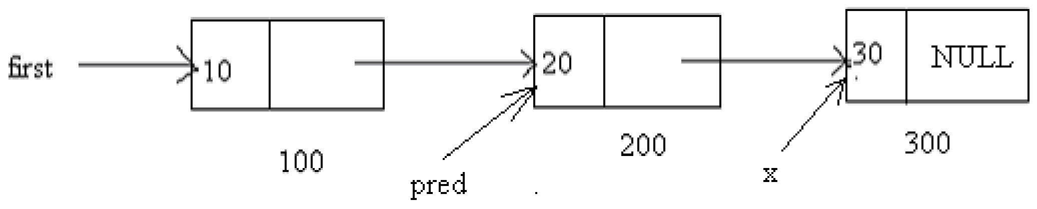


(b)

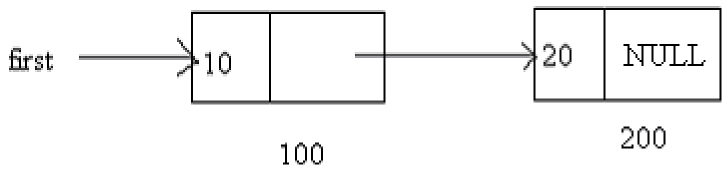


(c)

(iii) if x = 300 (Last node)



(a)



(b)

Traversing the List:

This includes, visiting the node and printing the data of that node one at a time and moving forward until it reaches end of the list. (i.e., link becomes NULL)

```
Procedure(first)
begin
    for (temp = first; temp!= NULL; temp = temp → link)
        write (temp → info)
end.
```

APPLICATIONS LINKED LISTS:

The main Applications of Linked Lists are

It means in addition/subtraction /multipication.. of two polynimials.

Eg:p1=2x^2+3x+7 and p2=3x^3+5x+2

p1+p2=3x^3+2x^2+8x+9

* In Dynamic Memory Management

In allocation and releasing memory at runtime.

*In Symbol Tables

in Balancing paranthesis

* Representing Sparse Matrix

IMPORTANT QUESTIONS UNIT - VIII

1. Declare a Circular Queue of integers such that F points to the front and R points to the Rear. Write functions
   (i) To insert an element into Queue
   (ii) To delete an element from Queue
2. Write a function in C to form a list containing the intersection of the elements of Two lists.
3. Write a C program for implementation of various operations on Circular Queue.
4. How can a polynomial in three variables (x, y and z) be represented by a single linked list? Each node should represent a term and should contain powers of x, y and z as well as coefficients of that term. Write a routine to evaluate this polynomial for given values of x, y and z.
5. Write a C program to convert a given prefix expression to postfix expression using stacks.
6. Write a non recursive simulation of Towers of Hanoi problem?

7. Write a routine to reverse elements of a doubly linked list by traversing the list only once.
8. Write in detail about the following
   (i) Recursion
   (ii) Applications of stacks and queues
9. Write a C program using pointers to implement Queue with all operations.
10. Write a program to convert a postfix expression to a fully paranthesized infix expression. For example AB+ would be transformed in to (A+B) and AB+C- would be transformed in to ((A+B)-C).
11. Define Linked list. Explain various operations performed on linked list with suitable Algorithms.
12. What is a Data structure? Explain the various types of data structures with suitable Examples.
13. What are advantages and disadvantages of stack? Write a program to illustrate stack Operation.
14. Define abstract data type. Explain with an example.
15. Give an algorithm / C program to reverse a singly linked circular list in place?
16. What is the difference between Queue and Circular queue. Explain about Circular queue operations.
17. Write a C program to insert and delete the elements from Circular doubly linked list.
18. Write a C program to evaluate the postfix expression.
19. Write a C program to implement a Singly linked list with all operations performed on it.
20. Declare a Queue of integers . Write functions
    (i) To insert an element in to queue
    (ii) To delete an element from queue
21. What is mean by linked list? What are the advantages and disadvantages in using linked list?

22. Write a C program to create a linear linked list interactively and to print the list and the total number of items in the list.
23. Write a C program using pointers to implement a stack with all the operations.
24. (a) What is a difference between linked list and an array?
    (b) Write a C program to reverse the elements in a singly linked list.