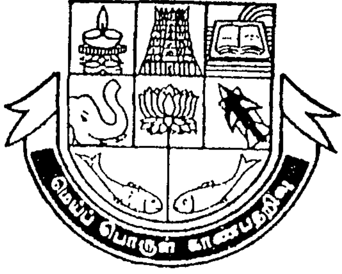


R 90

DIRECTORATE OF

DISTANCE EDUCATION

M
A
T
H
E
M
A
T
I
C
S

B.Sc., THIRD YEAR

PAPER – IX

PROGRAMMING IN C AND C++

Madurai Kamaraj University
Madurai – 625 021

All copyright privileges are reserved

Reprinted at SUN OFFSET, MADURAI.-16. ✆ 6549394 Yr. 2007-2008 Copies 2000

SYLLABUS
Major Paper - IX
Programming in C and C++

Unit I Introduction to Fundamental - Introduction to computer - Types of programming Languages - Introduction to C - The C character set - Identifiers and keywords - Data types -Constants - variables - Declaration - Expressions - various type of operators.

Unit II: Data input, output and control statements - Preliminaries single character input and output- Entering input data - writing output data - The gets and puts functions - Branching - Looping- Nested control structures - switch – Break - continue – Go to.

Unit III: **Functions** Overview - Defining a Function - Accessing a function - Function photo types - Passing Arguments to a function - Recursion - Library function Macros - The C preprocessor.

Unit IV: Storage classes - Automatic variables - Global variables - Static variables Register variables - Multiple programming - Bitwise Operation.

Unit V : **Arrays** Defining and processing of Array - passing Arrays Functions -Multi Dimensional Arrays - Arrays and Strings.

Unit VI : Pointers: Fundamentals - Declaration - passing pointers to function - Usage in one Dimensional and multi Dimensional Arrays - Dynamic Memory allocation Operations on pointers - Arrays of pointers passing functions to other Functions.

Unit VII **Structures and Unions:** Defining a structure - Processing a structure - Structures and pointers passing structures to Functions – Self referential structures - Bit Fields-Unions-Enumerations.

Unit VIII: Data Files: Opening and closing of Data Files - creating a data File - Processing a Data File - Unformatted Data File - command Line Parameter.

Unit IX: Queues: The Queues and its sequential representation - the quenue as an abstract Data Types - C implementation in queues - Priority queue.

Unit X: Linked List -inserting and moving Nodes from a list - Linked List as Data Structure - Examples of List operations - List operations - List implementation of priority Queues – Header Nodes - Lists in C - Linker list using Dynamic variables - Queues list in C - Implementing Header Nodes - Simulation using Linked Lists - other lists Structure.

Reference Books

1) Theory and Problems of programming with C By Byron S.Gottfried Ph.D,II Edition, Year 1998, The McGrow-Hill Publishing Chapters 1-4,6-14

2) Yedidyah Langsom, Moshe J.Augenstein and Aaron M.Tenanbaum "Data Structures using C and C++"

CONTENTS
Programming in C and C++

Unit No:	SCHEME OF LESSONS	Page.No
UNIT – 1:	**PROGRAMMING WITH C**	
1.	Introductory concepts	4
2.	Types of programming Languages	5
3.	The 'c' character set	6
4.	Identifiers	7
5.	Data types	8
6.	Integer constants	10
7.	Variables	12
8.	Declaration	13
9.	Expressions	13
10.	Various types of operators	14
Unit – 2:	**DATA INPUT, OUTPUT AND CONTROL STATEMENTS**	
1.	Preliminaries	21
2.	Single character input	21
3.	Single character output	22
4.	Entering Input Data	24
5.	Writing output Data	26
6.	The gets and puts functions	30
7.	Nested if	34
8.	Switch statement	36
9.	Nested switch statements	39
10.	While loop	44
11.	Continue statement	49
12.	Goto Label	50
Unit – 3:	**Functions**	
1.	Defining a function	53
2.	Calling function	54
3.	Categories of functions	55
4.	Passing Arguments to a function	61
5.	Recursion	63
6.	Macros and processors	65
7.	Library functions	67
Unit – 4:	**Storage classes**	
1.	Automatic variables	69
2.	Global variables	71
3.	Static variables	73

4.	Register variables	74
5.	Function in multiple programs	75
Unit – 5:	**Arrays**	
1.	Defining an array	79
2.	Passing an array to a function	82
3.	Two dimensional arrays	84
4.	Strings	89
Unit – 6:	**Pointers**	
1.	Introduction	96
2.	Pointer declaration	96
3.	Pointers and Functions	107
4.	Pointers and one Dimensional arrays	116
5.	Pointers and multidimensional arrays.	127
6.	Arrays of pointers	132
7.	Passing functions to other functions	139
8.	More about pointer Declaration	147
Unit – 7:	**Structures and unions**	
1.	Structures	149
2.	Initializing structures	151
3.	Structures and pointers	171
4.	Bit fields	178
5.	Enumerations	182
Unit – 8:	**Data files**	
1.	Opening closing of a Data file	185
2.	Creating a data file	187
3.	Procession a data file	191
4.	Unformatted data files	196
Unit – 9:	**Queues**	
1.	The queues and its sequential representation	201
2.	The queues as an abstract data type	202
3.	C- implementation of queues	203
4.	Priority queue.	207
Unit - 10	**Linked List**	
1.	Linked List	210
2.	Operation on Linked List	211
3.	List implementation of priority queues	217
4.	Header Nodes	217
5.	Lists in C – Array implementation of Lists	217
6.	Simulation using Linked Lists	219
7.	Other List structures	222

UNIT – 1
PROGRAMMING WITH C

1.1: Introductory concepts

Introduction to computers:

Today's computers come in many different forms. They range from massive, multipurpose mainframes and super computers to desktop – size personal computers. Between these extremes is a vast middle ground of minicomputers and work stations.

Mainframes and large minicomputers are used by many business. Universities, hospitals and government agencies to carry out sophisticated scientific and business calculations. These computers are expensive and may require a sizeable staff of supporting personnel and a special, carefully controlled environment.

Personal computers are small and inexpensive. Infact, portable, battery-powered "laptop" computers weighting less than 5 or 6 seconds are now widely used by many students and traveling professionals. Personals computers are used extensively in most schools and business and they are rapidly becoming common house hold items. Most students use personal computers when learning to program with c.

Introduction to C:

'C' seems a strange name for programming language. This strange sounding language is one of the most popular computer languages today. C was an off spring of the "Basic combined programming language" (BCPL) called B developed in 1960's at Cambridge university. B language was modified by Dennis Ritchie and was implemented at Bell laboratories in 1972. The new language was named C. It was developed along with "Unix" operating system it is strongly associated with 'Unix'.

'C' was used in academic environments, but eventually with the release of C compiles for commercial use and the increasing popularity of UNIX. Today, C is running under a number of operating systems including MS – Dos. Since MS-Dos is a dominant operating system for microcomputers. C has begun to influence the microcomputers community at large.

Importance of C:

C is a robust language whose rich set of built in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of high level language.

Programs written in C are efficient and fast. This is due to variety of data types and powerful operators. It is many times faster than BASIC. There are only 32 keywords and its strength lies inits bolt in functions. C is highly portable. This means that C programs written for one computer can be run on another with little on no modification portability is important if we can plan to use a new computer with different operating system.

Structure of C program:

Every C program consists of one or more modules called functions. One of the functions must be called main. The program will always begin by executing the main function, which may access other functions.

Each function must contain:

(i) A function heading, which consists of the function name followed by an optional list of arguments enclosed in parentheses.

(ii) A list of argument declarations, if arguments are included in the heading.

(iii) A compound statement which comprises the remainder of function.

The arguments are symbols that represent information being passed between the function and other parts of program.

Each compound statement is enclosed within a pair of braces (ie) { }. The expression may contain one or more elementary statements and other compound statements. These compound statements may be nested one within another. Each expression statement must end with a semicolon(;) comments may appear anywhere with in the program, as they are placed with in the delimeters /* & */. Such comments are helpful in identifying the program's principal features.

1.2: Types of programming languages

There are many different languages can be used to program a computer. The most basic of these is **machine language** – a collection of cryptic instructions that control the computer's Internal circuitry. Very few computer programs are actually written in machine language, however, for two significant reasons: First, because machine language is very cumbersome to work with and second, because every different type of computer has its own unique instruction set.

As a rule, a single instructions in a **High level language** will be equivalent to several instructions in Machine language. This greatly simplifies the task of writing complete, correct programs. The rules for programming in a particular high level languages are much the same for all computers, so that a program written for one computer can generally be run an many different computers with little or no alteration.

A program that is written in a high level language must, however be translated into machine language before it can be executed. This is known as compilation or interpolation.

A compiler or interpreter is itself a computer program. It accepts a program written in high level language (Ex.c) as input, and generates a corresponding machine-language program as output. The original high –level program is called **Source program** and the object program.

1.3 :The 'C' character set:

The characters are used to form words, numbers and expressions, in any programming language. These depend upon individual style of each programming language.

(a) Letters:

Consists uppercase letters A to Z and lowercase letters a to z . It shall be noted that upper and lower case letters are not interchangeable, as c is strongly case sensitive.

(b) Digits:

Cc·.sists numbers from 0 to 9.

(c) Special characters:

(i) , → comma
(ii) ; → Semicolon
(iii) : → Colon
(iv) ? → Question mark
(v) ' → Single quotation
(vi) " → Double quotation

Viii) ! ≯ exclamation mark
Viii) | ≯ vertical bar
ix) / ≯ slash
x) \ ≯ back slash
& ≯ ampersand
^ ≯ caret
* ≯ asterick
( )≯ left & right parentheses

xi) ~ ⇒ tilde

xii) – ⇒ underscore

xiii) % ⇒ percent sign

⇒ hash

$ ⇒ dollar sign

{ } ⇒ left right braces

(d) White spaces: Consists of blank space, newline, form feed carriage return and horizontal tab.

C use certain combinations of these characters as \b, \n & \t to represent special conditions such as backspace, newline, and horizontal tab. These character combination are known as **escape sequences**.

1.4: Identifiers:-

Identifiers are names that are given to various program elements such as variables, functions and arrays.

1) Identifiers consists of letters and digits, in any order, except the first character must be a letter.
2) Both upper case and lower case letters are permitted, through common usage favours the use of lower case letters for most type of identifiers.
3) Upper case and lowercase letters are not interchangeable.
4) The under score (–) can also be included and it is considered to be a letter.
5) An Identifier may also begin with an underscore.

Examples:-

The following names are **valid** Identifiers

(i) x (ii) y12 (iii) Sum_1

(iv) _temperature (v) TABLE (vi) area

The following names are not valid identifiers for the reasons stated.

	Names		Reasons
(i)	4th	⇒	The first character must be a letter
(ii)	" x "	⇒	Illegal character (")
(iii)	error flag	⇒	Illegal character (blank space)

Key words:-

There are certain reserved words, called keywords.

i) All keywords have fixed meanings and these meanings cannot be changed.
ii) Keywords serve as basic building blocks for program statements.
iii) All keywords must be written in lower case.

The standard keyword are

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

1.5: Data types:

c language is rich units data types. Storage representations and machine Instructions to handle constants differ from machine to machine. The variety of datatypes available allow the programmer to select the type appropriate to needs of application as well as the machine.

C supports four classes of datatypes:
(1) Primary data types
(2) User – defined types
(3) Derived data types
(4) Empty data set

The primary datatypes and their extensions are discussed in this section. The user – defined datatypes are discussed in next section.

Derived datatypes such as arrays, functions, structures and pointers are discussed as when they are encountered. The empty data set is discussed in the chapter on functions.

All C compilers support four fundamental datatypes. There are
(i) Integer (int)
(ii) Character (Char)
(iii) Floating point (float)

(iv) Double – precision floating point (double) many of them also extended datatypes such as long int and long double.

Primary data types			
Integral type			
Integer		**character**	
Signed type	Unsigned type	Signed char	
Int	Unsigned int	UnSigned char	
Short int	Unsigned short int		
Long int	Unsigned log int		
Floating Point Type			
float	double	long double	

Size and Range of basic Data types

Datatypes	Range of values
char	– 128 to 127
int	– 32,768 to 32,767
float	3.4e – 38 to 3.4e +38
double	1.7e – 308 to 1.7e +308

Integer Types:

Integers are whole numbers, the size of an integer value can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is –32768 to +32767.

A signed integer used one bit for sign. In order to provide some central over the range of numbers and storage space, c has three classes of integer storage.

1. Short int
2. int
3. long int

Short int represents small integer values and requires half the amount of storage as regular int number uses. Unlike signed integers, **Unsigned integers** use *all* the bits for magnitude of number and are always positive. We declare long and unsigned integers to increase the range of values.

Floating point Types:

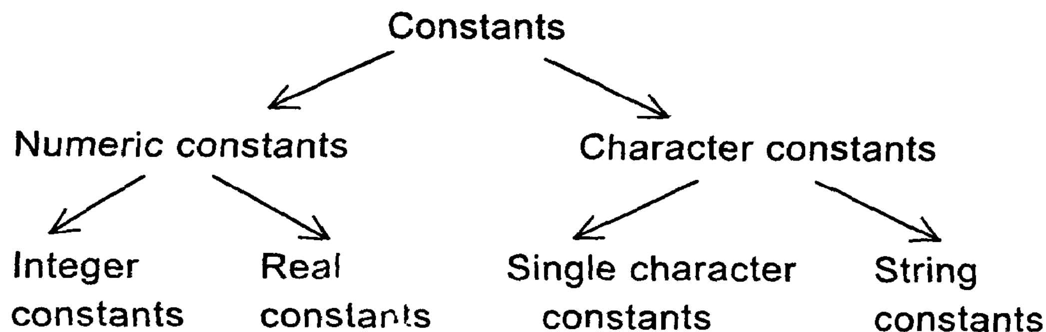
Floating point numbers are defined in c by the keyword **float**. When the accuracy provided by a float number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. To extend the precision. We may use **long double** which uses 80 bits.

Character Types:-

A single character can be defined as *character(char)* type data. Characters are usually stored in 8 bits of internal storage. While un signed chars have values between 0 & 255. Signed chars have values from –128 to 127.

Constants:-

There are four basic types of constants in C.



(B)

1.6 Integer constants:-

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

(i) Decimal integers consists of a set of digits 'o' through '9' preceded by an optional – or + sign. Spaces, commas and non – digit characters are not permitted between digits.

Vaid ex	123,	– 321,	0,	+78
Invalid ex:-	15750	20,000	,	$1000

(ii) An octal integer constant consists of any combination of digits frooom 0 through 7, with a leading 0.

Eg: - Ox2, O_x9F, Oxbcd

(iii) A sequence of digits preceded by O_x or OX is considered as hexadecimal integer. They may also include alphabets A through F or f. The letters A through F represent the numbers 10 through 15.

Real constants:-

Integer numbers are in adequate to represent quantities that very continuously such as distances, heights, temperature etc. These quantities are represented by numbers containing fractional part like 17.54. Such numbers are called real or floating points constants. A real number may also be expressed in exponential notation.

Ex: 0.0083, +247.5

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign

Ex: $0.65e^4$, 1.5e+5, – 1.2E – 1

Single character constants:–

A single character constant contains a single character enclosed with in a pair of single quote marks.

Ex: '5', 'x' ' ; ' "

String constants:–

A string constant is a sequence of characters may be letters; numbers, special characters and blank space.

Ex:- " Hello ", "1987 ", " 5+3 "

A character constant is not equivalent to single character string constant.

Escape sequences:-

Certain nonprinting characters as well as the back slash (\) and the apostrophe (') can be expressed in terms of escape sequences. An **escape sequence** always begins with a backward slash and is followed by one or more special characters.

Character	Escape sequence	ASCII value
bell (alert)	\ a	007
backspace	\ b	008
horizontal tab	\ t	009

vertical tab	\ v	011
new line	\ n	010
formed feed	\ f	012
carriage return	\ r	013
quotation mark (")	\ "	034
apostrophe (')	\ '	039
question mark	\ \	092
null	\o	000

1.7: Variables:

A variable is an identifier that is used to represent a single data item (ie) a numerical quantity or a character constant. The data item must be assigned to the variable at same point of the program. The data item can then be accesses later in the program simply by referring to variable name. A given variable can be assigned different data items at various places with in the program. The information represented by the variable can change during the execution of program. The data type associated with the variable cannot change.

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature of program. Variables names may consist of letters digits and underscore (–)

Rules:-

(i) They must begin with a letter.
(ii) ANSI std recognizes a length of 31 characters. However the length should not be normally more than eight characters.
(iii) Upper case and lower case are significant.
(iv) The variable name should not be a keyword.
(v) White space is not allowed.

Valid examples:-

First_tag, average, value, sum1

Invalid examples:-

Variable name				Reasons
Char	→	Notvalid	→	Char is keyword
Price $	→	Notvalid	→	Dollar sign is illegal
Group one	→	Notvalid	→	Blank space is not permitted

Ex: A C Program contains following lines

```
Int a, b, c;
Char d;
----------
```

```
a = 3;
b = 5;
c = a + b;
```

1.8 Declarations:-

A **declaration** associate a group of variables with specific data type. All variables must be declared before they can appear in executable statements.

A declaration consists of a data type followed by one or more variables names, ending with a semicolon.

Integer type variables can be declared to be short integer for smaller integer quantities or long integer for larger integer quantities. Such variables are declared by writing short int & long int.

An integer variable can also be declared to be unsigned by writing by writing unsigned int **floating point** variables can be declared to be double precision by using the type double or long float rather than float.

Initial values can be assigned to variables within a type declaration. The declaration must consists of data type, followed by a variable name, an equal sign (=) and a constant.

Examples:

```
int a;
short int a;
unsigned int a;
double a;
float a;
int a = 12;
```

1.9: Expressions:-

An expression represents a single data item, such as a number or a character. The expression may consist of a single entity, such as constant, a variable an array element or reference to a function. It may also consists of some combination of such entities, inter connected by one or more operators.

Expressions can also represent logical conditions that are either true or false. In C, the conditions true or false are represented by integer values 1 and 0. Hence logical type expression really represent numerical quantities.

Examples:

```
a + b
x = y
C = a + b
x = y
```

1.10: Various Types Of Operators:

An **operator** is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables.

C operators can be classified into number of categories. They are

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

Arithmetic Operators:

C provides all the basic arithmetic operators. The operators +, —, * and / all work the same way as they do in other languages,

The Unary minus operator, in effect multiplies its single operand by —1. Therefore a number preceded by a minus sign changes its sign.

Operators	Meaning
+	Addition or Unary plus
—	Subtraction or Unary minus
*	Multiplication
/	Division
%	Modulo division

Integer division truncates any fractional part. The Modulo division produces the remainder of an integer division.

Examples:

a + b	a – b
a * b	a / b
a % b	–a * b

Here a and b are variables and are known as operands.

Integer Arithmetic:

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

Example: If a = 14 & b = 4

We have the following results

a + b = 18
a – b = 10
a + b = 56
a / 3 = 3 (decimal part truncated)
a % b = 2 (remainder of division)

Real Arithmetic:

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

Example:

x = 6.0 / 7.0 = 0.857143
y = –2.0 / 3.0 = – 0.666667

The operator % cannot be used with real operands

Mixed – mode Arithmetic:

When one of the operands is real and other is integer, the expression is called a mixed–mode arithmetic expression.

Example:

(1) 15 / 10.0 = 1.5
(2) 15 / 10 = 1

Relational operators:

If we want to compare the age of two persons, or the price of two items and so on. These comparisons can be done with the help of relational operators/

An expression such as

a < b or 1 < 20 containing a relational operator is termed as relational expression. The value of a relational expression is either one or zero. It is one if the specified rotation is true and zero if the relation is false.

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

Examples:

4.5   < =   10   → TRUE
4.5   <   –10   → FALSE
10   <   7 + 5   → TRUE

Logical operators:

C has the following three logical operators.

& & → AND
| |
| | → OR
! → NOT

The logical operators & & and ! ! are used when we want to test more than one condition and make decision.

Example: a > b & & x = = 10

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

The logical expression given above is true only if a < b is true and x = 10 is true. If either or both of them are false, the expression is false.

Assignment operators:

Assignment operators are used to assign the result of an expression to a variable. The Usual assignment operator is ` = '. In addition C has a set of `Short hand' assignment operators of the form.

VOP = exp;

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

The assignment statement is

VOP = exp;

is equivalent to

V = V op (exp);

Example:

x + = y + 1 ;

This is same as statement

x = x + (y+1);

Statement with Simple assignment operator	Statement with short hand operator
a = a + 1	a + = 1
a = a – 1	a – = 1
a = a * (n +1)	a * = n + 1
a = a / (n +1)	a / = n + 1
a = a % b	a% = b

Increment and Decrement operators:

C has two very useful operators. They are ++ and – –.

The operator ++ adds 1 to the operand while – – subtracts 1. Both are Unary operators and the following form:

++ m or m++;

–– m or m––;

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

Consider m = 5;

y = ++m;

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

m = 5;

y = m++;

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

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left.

A post fix operator first assigns the value to variable on left and then increments the operand.

Conditional operator:

The conditional expressions of the form.

exp 1 ? exp 2: exp 3;

The operator?: works are

(i) exp 1 is evaluated first

(ii) If it is true then the expression exp 2 is evaluated and becomes the value of expression.

(iii) If exp 1 is false, exp 3 is evaluated and its value becomes the value of expression.

Example:

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

In the above example exp 1 is false the answer is exp 3 (ie) x = 15 will be the value.

Bit wise operators:

These operators are used for testing the bits of shifting them right or left. Bit wise operators may not be applied to float or double.

The bit wise operators and their meanings are given below.

Operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
> >	Shift right
< <	Shift left
~	one's complement

Special operators:

Special operators are comma operator, size of operator, pointer operators (& and *) Member selection operators (. and →).

We will see comma and size of operators. The pointer operators are discussed later. Member selection operators are used to select members of structure.

The Comma operator:

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

Example: value = (x = 10, y =5, x + y);

First assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value.

Comma operator has lowest precedence of all operators.

The Size of operators:

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

Example:

```
m =  size of (sun);
 n =  size of (long int);
 k = size of (235 L);
```

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

Now we see one program by using all the operators.

Program:

```
        / * Illustration of operators * /.

Main ()
{
        int a, b, c, d;
        a = 15;
        b = 10;
        c = ++a – b;
```

```
Print f ("a = % d       b = % d         c = % d (n," a, b, c);
        d = b ++  +a;

Print f ("a = % d       b = % d         d = % d \n", a, b,d);
Print f ("a/b =         % d \n",        a / b);
Print f ("a%%b =        % d \n",        a% b);
Print f ("a * = b =     % d \n",        a * = b);
Print f ("% d \n",      (c > d)? 1 : 0);
Print f ("% d \n",      (c < d) ? 1 :0);
}
```

Output:

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

UNIT – 2
DATA INPUT AND OUTPUT

2.1 Preliminaries:

An input | output function can be accessed from anywhere within a program simply by writing the function name, followed by list of arguments enclosed in parentheses.

The arguments represent data items that are sent to the function. Some input | output functions do not require arguments, though the empty parentheses must still appear.

The names of those functions that return data items may appear within the expressions, as though each function reference were an ordinary variable (Ex: C = get char ();) or they may be referenced as separate statements (Ex: Scanf (.....) ;) Some functions do not return any data items such functions are referenced as though they were separate statements (Ex: put char (...);)

C include a collection of header files that provide necessary information in support of library functions. Each file generally contains information in support of a group of related library functions. These files are entered into the program via an # include statement at the beginning of the program. The header file required by the standard input | output library functions is called stdio.h (ie) # include < stdio,h >

2.2 Single Character Input:

The get char Function

Single character can be entered into the computer using the C library function get char.

The get char function is a part of the standard C I/O library. It returns a single character from a standard input device (typically a key board). The function does not require any arguments, though a pair of empty parentheses must follow the word get char.

The get char takes the following form.

Character Variable = get char ();

Where Character variable refers to some previously declared character variable when this statement is encountered, the computer waits until a key pressed and then assigns this character as a value to get char function. Since get

char is used on the right hand side of an assignment statement, the character value of get char is turn assigned to the variable name on left.

For example

```
Char name;
name = get char ();
        Or
Char name = get char ();
```

We will assign the character `H' to the variable name when we press the I on the key board. Since get char is a function requires a set of parentheses.

Example program:

```
/ * Reading a character * /
# include < stdio. h >
main ()
{
   Char answer:
   printf ("would you like to know my name?".);
   printf ("Type Y for yes and N for No");
   Answer = get char ();
If (answer = = `Y' || answer = = `Y')

   printf ("\n my name is Busy Bee \n");
else
   printf ("\n you are good for nothing \n");
}
```

Output:

```
Would you like to known my name?
Type Y for yes and N for No : Y
My name if Busy Bee
Would you like to know my name?
Type Y for yes and N for No: N
You are good for nothing.
```

2.3 Single character output

The Put char Function:

Single characters can be displayed (i.e written out of the computer). Using the C library function put char.

This function is complementary to the character input function get char. The put char function, like get char, is a part of standard C I/O library. It transmits a single character to a standard output device (typically a TV monitor).

The character being transmitted will normally be represented as a character – type variable. It must be expressed as an argument to the function, enclosed in parentheses, following the word put char.

The put char function is of the form put char (character variable)

Example:

```
answer: `Y'
put char (answer);
```

It will display the character Y on the screen.

Program:

The following program uses three new functions.: is lower, to upper and to lower. The function is lower is a conditional function and takes the value TRUE if the argument is lower case alphabet, otherwise takes the value FALSE. The function to upper converts the lower case argument into an upper case alphabet while the function to lower does the reverse.

```
/ *  WRITING THE CHARACTER  * /
# include < stdio.h >
# include < ctype.h >
main ()
{
   char alphabet
   printf ("Enter an alphabet");
   Put char (`\n')
   alphabet = get char ();
   if (is lower (alphabet));
   Put char (to upper (alphabet));
else
   put char (to lower (alphabet));
}
```

Output:

```
        Enter an alphabet
        a
        A
Enter an alphabet
        Q
        q
```

The Put char function can be used to output a string constant by storing the string within a one–dimensional character–type array.

2.4 Entering Input Data:

The Scanf function:

Input data can be entered into the computer from a standard input device by means of C library function Scanf.

This function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully.

The Scanf function can be written as

format:

Scanf (control string, arg1, arg2, ... argn);

Control string specifies field format in which the data is to be entered and the arguments arg1, arg2 . . . . argn specify the address of location where the data is stored. Control string and arguments are separated by commas.

Control string contains field specifications which direct the interpretation of input data.

(i) Field specifications consisting of the conversion character % of data type character and an optional number, specifying the field width

(ii) Blank, tabs or new lines

Within the control string, multiple groups can be contiguous or they can be separated by white space characters. If white space characters are used to separate multiple character groups in the control string, then all consecutive white space characters in the input data will be read but ignored.

Conversion Character	Meaning
c	data item is a single character
d	data item is a decimal integer
f	data item is a floating point value
h	data item is a short integer
i	data item is decimal, hexadecimal or octal integer
o	data item is octal integer
s	data item is string followed by a white space character

The arguments are written as variables or arrays, whose types match the corresponding character groups in the control string. Each variable name must be preceded by an amper sand (&) Array names should not begin with an amper sand.

Example:

```
Scanf ("%d %d", & No1, & No2);
```

will read the data.

314 50

correctly assign 314 to No 1, and 50 to No 2

An Input field may be skipped by specifying in the place of field width.

for Example:

```
Scanf ("%d % *d%d", &a &b);
```

will assign the data

12 45 78

12 to a
45 skipped (because of *)
78 to b

Example:

```
Scanf ("% wd", & c);
Scanf ("%3d", & c);
```

will assign the data

123

w represents the width of the Number

2.5 Writing Output Data:

The printf function:

Output data can be written from the computer onto a standard output device using library function printf. This function can be used to output any combination of numerical values, single characters and strings.

The printf function moves data from the computers memory to the standard output device, whereas the Scanf function enters data from the standard input device and stores it in the computers memory.

The printf function is written as

Format: printf (control string, arg1, arg2 . . . . argn);

Control string consists of three types of items

(i) characters that will be printed on the screen as they appear.

(ii) Format specifications that define the output formed for display of each item

(iii) Escape sequence characters such as \n, \t and \b.

The arguments arg1, arg2 . . . . argn are the variables whose values are formatted and printed according to the specifications of the control string. The arguments should match in number.

The control string consists of individual groups of characters, with one character group for each output data item. Each character group must being with percent sign (%). An individual character group will consist of percent sign, followed by conversion character `indicating the type of corresponding data item'.

Multiple character groups can be contiguous or they can be separated by other characters, including white space characters.

A minimum field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field.

Examples:

1) printf ("%d", a); for integer Numbers
2) printf ("%f", a); for float Numbers
3) printf ("welcome to c"); for strings
   Output will be
      Welcome to C.
4) printf ("%4d", 1076);
   The output is
      1076
5) printf ("%6d", 1076);
   The output is
      6b 1076
   b is blank space
6) printf ("%–6d", 1076)
   The output is
      1076 6b → blankspace
   Here the minus (–) sign is used for left–justified printing
7) If a = 10.77682
   printf ("%7.4f",a);
   The output is
      10.7768
8) printf ("%7.2f", a);
   The output is
      10.77

Programs: Using data type, variables Scanf and printf

```
/ * To find the area of a triangle * /
# include < stdio.h >
main ()
{
   float area, base,ht;
   printf ("enter b,h\n");
   Scanf ("%f%f", & base, &ht);
   area = 0.5 * base * ht;
   printf ("area = %f\n", area);
   }
```

Output:

```
Enter  b, h
15.0   10.0
area = 75.0000
```

Explanation for above program:

Line 1: Comment line to explain the purpose of the program

Line 2: To open the standard input/output library.

Line 3: It is costmary in C all the programs to start with a function called main ().

Line 4: *

Line 5: Declaration of floating variables a,b,n

Line 6: printf is used to print the message “enter b,h”.

Line 7: Scanf is used to accept values for b,h from uses.

Line 8: To compute area

Line 9: To display the result

Line 10: *

* Here the braces (line No 4 & 10) are used to indicate the beginning and end of function main ().

Program:

```
/ * program to calculate simple interest * /
# include < stdio.h >
main ()
{
   float  prin, noy, roi, intrst;
   printf (“%f”, & prin);
   Scanf (“%f”, & prin);
   printf (“enter number of years\n”);
   Scanf (“%f”, & noy);
   printf (“enter rate of interest \n”);
   Scanf (“%f”, & roi);
   intrst = (prin * noy * roi) /100;
   printf (“interest %7.2f”,intrst);
}
```

Output:

```
enter principal
1000
enter Number of years
5
enter rate of interest
15
interest 750.00
```

Program:

```
/ * program to find centigrade for a
given Fahrenheit temperature * /
# include < stdio.h >
main ()
{
  flood centi, fahren;
  printf ("enter Fahrenheit temperature \n");
  Scanf ("%f", & fahren);
  Centi = 5.0 /9.0 * (fahren – 32);
  printf ("\n the centigrade temp is % 6.2f", centi);
}
```

Output:

```
enter fahrenheit temperature
100
the centigrade temp is 37.78
```

Program:

write a program to reverse a given 5 digit integer number

```
/ * program to reverse a given 5 digit number* /
main ()
{
   int a,b,c,d,e,No;
   printf ("\n enter number to reverse \n");
   Scanf ("%d", & No);
   a = No % 10;
   b = (No /10)%10;
   c = (No/100)%10;
   d = (No /1000)%10;
   e = No /10000;
   printf ("\n Reversed Number");
   printf ("%d%d%d%d%d",a,b,c,d,e);
}
```

Output:

```
enter Number to reverse
12345
Reversed Number is 54321
```

2.6: The gets and puts Functions:

C contains a number of other library functions that permit some form of data transfer into or out of the computer.

The gets and puts functions, which facilitate the transfer of strings between the computer and the standard input /output devices

Each of these functions accepts a single argument. The argument must be a data item that represents a string (Ex: character array). The string may include white space characters.

In the function gets, the string will be entered from keyboard and will terminate with a new line character.

The gets and puts functions offer simple alternatives to the use of scanf and printf for reading and displaying strings

Example:

Write a program to read and write the line of text using gets & puts.

Program:

```
/*Reading and writing the line of text*/

#include <stdio.h>
main ()
{
    char line[80];
   gets (line);
   puts(line);
}
```

CONDITIONAL STATEMENTS:

C supports two types of conditional statements

1. if 2.Switch

If is classified as simple if, multiple if and Nested if.

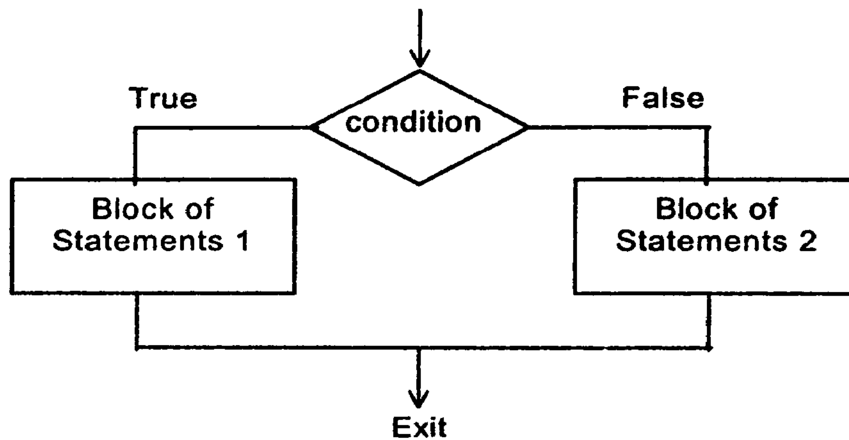
Simple if:-

Using simple if you can check for single condition only.

Syntax:-

If (condition)

```
{
    block of statements1;
}
else
{
    block of statements2;
}
```



If the condition is true then Block of statements 1 is executed. Other wise block of statements 2 is executed (if it exists). Else part is optional.

Program:-

```
# include <stdio.h>
main()
{
   int  magic _no=123;
   int guess;
   printf("Enter your guess:");
   Scanf ("%d", & guess);
   If (guess==magic_no)
   {
      printf("You guessed right");
   }
}
```

The above program uses the equality operator to determine whether your guess matches the magic_no. If it does, the message "you guessed right" is displayed on the screen. If it does, the message "you guessed right" is displayed on the screen. If it does not the match the magic_no, no operation is performed.

Above program with a slight improvement:-

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

```
        int magic_no = 123
        int guess;
        printf("Enter you guess:");
        scanf("%d",& guess);
        if(guess==magic_no)
       {
          printf("you guessed  right");
       }
    else
    {
        printf("you guessed wrong");
       }
    }
```

As is clear from the above program, if the guess equals the magic_no, the code associated with the if statement is executed, otherwise the code associated with the else statement is executed.

Program to check whether given year is leap year or not:-

```
#include<stdio.h>
main()
{
   int yy;
   printf("Enter the year:");
   scanf("%d",&yy);
   if((yy%4==0 && yy%100!=0) !!(yy%400==0))
   {
       printf("LEAP YEAR");
}
else
{
     printf("NOT A LEAP YEAR");
   }
}
```

Multiple ifs:-

Using simple if you can check for more than one condition.

Syntax:-

```
If(condition)
{
   block of statements;
```

```
}
else if(condition)
{
   block of statements;
}
else if (condition)
{
   block of statements;
}
:
else
{
   block of statements;
}
```

The conditions are evaluated from top to bottom. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is by passed. If none of the conditions are true, the final else is executed. If no final else is present, then no action takes place if all other conditions are false.

Example for multiple ifs:

Program:

```
# include<stdio.h>
main()
{
   int magic_no=123;
   int guess;
   printf("Enter your guess");
   if(guess==magic_no)
   {
   printf("You guessed right");
   printf("The magic number is %d", magic_no);
}
else if (guess>magic_no)
   {
     printf("wrong: the number is too high");
    }
else
   {
    printf("wrong: the number is too low");
     }
}
```

Program:-

```
#include<stdio.h>
main()
{
    int input_char;
     printf("Enter a character:");
     input_char=getchar();
      if(input_char>='a'&&input_char<='z')
       {
      printf("lowercase character\n");
}
else
    if(input_char>='A' && input_char<='z')
      {
       printf("upper case character\n");
       }
     if(input_char>='0' && input_char<='9')
       {
       printf("digit\n");
        }
else
  {
   print("special character\n");
  }
}
```

2.7 Nested if:-

Using Nested if at a same time you can check for more than one condition

Syntax:-

```
If(condition)
{
   if(condition)
   {
   block of statements;
    }
 else
   {
    block of statements;
   }
}
```

The outer if conditions is evaluated first if it returns true then the contro passes on to the if, if the inner if condition is also true then the consequent block of statements is executed or if the inner if condition evaluated to false then the inner else part is executed. If the outer if condition is evaluated to false then the control jumps of if statement, the control never passes on to inner if.

Example for nested if:

Program:

```
#include<stdio.h>
main()
{
    char ch ='yes';
    int temp;
    printf("what is the tempeıature outside?");
    scanf("%d",&temp);
    printf("Is it raining outside?");
    scanf("%c",ch);
    if(reply=='yes')
    if(temp<25)
    printf("put on your rain coat");
  else
    printf("Take the umbrella with you");
  else
   if(temp<25)
   printf("put on your overcoat");
 else
   printf("put on your jacket");
}
```

The reason that nested ifs are trouble some is that it can be difficult to know what else associates with what if. Fortunately, c provides a very simple rule for resolving this type of situation. In c, the else is linked to the closest proceeding if that does not already have an else statement associated with it.

Program:-

```
#include<stidio.h>
main()
{
   int magic_no=123;
   int guess;
   printf("Enter your guess:");
   scanf("%d",&guess);
```

```
if(guess==magic_no)
{
printf("you guessed right");
printf("The magic number is %d",magic_no);
}
else
{
  printf("Wrong");
  if(guess>magic_no)
  printf("The number is too high");
else
  printf("The number is too low");
 }
}
```

2.8 Switch Statement:

Sometimes, the use of multiple if statements becomes difficult to comprehend. As an alterative, built in multiple branch decision statement, **Switch** can be used to find a match with the given condition. A variable is successfully tested against a list of integer or character constants. When a match is found, a statement or block of statements is executed.

Syntax:-

```
Switch(variable)
{
case constant1:
        statement sequences;
  break;
case constant 2:
        state. .ent sequences;
   break;
:
case constant:
        statement sequences;
      break;
  default:
     statement sequences;
}
```

where the default statement is executed if no matches are found. The default is optional and, if not present, no action takes place if all matches fail. When a

match is found, the statement associated with that case is executed until the break statement is reached.

There are 3 things to be kept in mind in case of **switch** statement:

1. The switch statement can only test for constants and not expressions or strings.
2. No two **case** constants in the same switch statement can hare identical values
3. More than one **case** can have same block of statements.
4. If no **break** statement is not provided for each and every **case** then all the case is executed even if the condition gets satisfied at the first case statement. So it is mandatory to provide **break** in each and every case.

Program:-

To carry out various arithmetic operation on two numbers:-

```
#include<stdio.h>
main()
{
  float num1,num2;
  float ans;
  char operator;
  printf ("Enter the first number:");
  scanf("%f",&num1);
  printf("Enter the second number:");
  scanf("%f",&num2);
  printf("Enter the operator:");
   scanf("%s",&operator:);
switch(operator)
{
   case '+': ans = num1 + num2;
                break;
   case '–': ans = num1 + num2;
                break;
   case '*': ans = num1 + num2;
                break;
   case '/': ans = num1 + num2;
                break;
         if(num 2! = 0.0)
         {
            ans = num1 / num2 ;
```

```
                break;
        default: "Invalid operator entered";
    }
        printf("The result is %f", ans);
}
```

Output:-

```
Enter the first number: 20
Enter the Second number: 5
Enter the operator: $
Invalid operator entered
```

Program to find out whether a given year is a leap year.

```
# include<stdio.h>
main()
{
    int d,m,y;
    int no_of_days;
    printf("Input a data in the format DD MM 19YY\n");
    scanf("%d%d%d",&d,&m,&y);
    switch(m)
{
    case  1:
    case  3:
    case  5:
    case  7:
    case  8:
    case 10:
    case 12:
         no_of_days = 31;
         break;
     case 4:
     case 6:
     case 9:
    case 11:
         no_of_days = 31;
         break;

case 2:
         if(y - %4 ==0)
         {
            no_of_days = 29;
```

```
printf("%d is a leap year", y);
 }
else
{
  no_of_days = 28;
  printf("%d is not a leap year", y);
}
break;
default : "Invalid month entered";
}
}
```

Out put :

Input a data in the format DD MM 19YY

12 3 1997

1997 is not a leap year.

The execution continues in the next case if there is no break statement sequence as in:

```
switch(c)
{
  case 1;
     int;     /* this is incorrect*/
  :
```

But a variable can be added as shown here

```
Switch(c)
 {
  int i;
  case 1:
  :
```

2.9: Nested switch statement:-

It is possible to nest switch statements within other switch statements. The case constants of the inner and outer switch statements can code fragment is perfectly accepted :-

```
Switch (x)
{
    case 1:
       switch (y)
       {
```

```
case 1: pow (x,y);
      break;
default: printf("This is a nested switch example");
      break;
 case 2:
   ⋮
```

Looping constructs:-

Loops allow a set of instructions to be performed until a certain condition is reached. There are three types of loop constructs in c:-

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

for loop:-

syntax:-

```
for (initialization; condition; increment)
{
   statements;
}
```

The for loop has 3 main parts:-

1. Initialization is an assignment statement that is used to set initial value of the loop control variable.
2. Condition is a relation expression that must be true for the loop to continue execution. It also determines when the loop will exit.
3. Increment defines how the loop control variable will change each time the loop is repeated.

The **for** loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for** loop.

Program prints numbers from 1 to 20 on the screen:-

```
#include<stdio.h>
main()
{
  int x;
  for(x=1;x>=20;x++)
   {
    printf("%d",x);
    }
  }
```

In this program, x is initially set to 1. Since the value of x is less than 20, printf() is called, x is increased by 1, and x is tested to see if it is still less than or equal to 20. This process repeats until x is greater than 20, at which point the loop terminates. In this example, x is the loop control variable, which is changed and checked each time the loop repeats.

An important point about for loops is that the conditional test is always performed at the top of the loop. This means that he body of the loop may not be executed at all if the condition is false to begin with. Another point is that the increment is evaluated at the end of each pass of the loop.

The third section of the for statement is some what misleading. Increment can not only be used to increase the value of the loop

Control variable, but it can also be used to decrease the value of the control variable.

Program:-

```
# include<stdio.h>
main( )
{
    int x;
    for (x=20; x>=1; x – –)
    {
        printf (" %d ", x);
    }
}
```

Here the initial value of x has been set to 20. The value of x is gradually decreased by 1. Each value is tested against the condition x >= 1. If condition is true, the number is displayed in reverse order from 20 to 1. When the value of x becomes lesser than 1, the condition fails and the loop terminates **for** statements are generally suited for loops in which the number of passes is known in advance.

For loop variations:–

Several variations are allowed that increases the power, flexibility, and applicability to certain programming situations. One of the common variations is achieved by using more than one variable to control the loop. For example, this loop uses two variables x and y to control the loop, with both the variables being initialized inside the **for** statement:-

Program:-

```
# include<stdio.h>
main()
{
    int x,y;
    for (x =1, y = 0; x < 20; x++)
{
    y = y+x;
    printf (" %d %d %f \n", x,y,p);
}
}
```

Here the two initialization variables are separated by a comma separator.

Consider another example, in which more than one variable is used in the initialization as well as increment section of the **for** statement.

Program:-

```
# include<stdio.h>
main()
{
    int x,y;
    float p;
    for(x=1,y=50;x<20;x++, y– –)
    {
        p=y / x;
        printf("%d %d %f \n",x,y,p);
    }
}
```

Another feature is that the condition may have any compound relation and the testing need not be limited only to the loop control variable. Consider the example below:-

Program:-

```
# include<stdio.h>
main ()
{
    int x,sum;
    for(x=1; (x<20 && sum < 100); x++)
    {
        sum=sum+x;
        printf("%d %d \n", x,sum);
    }
}
```

The loop uses a compound test condition with the control variable x and the external variable sum. The loop is executed as long as the condition x<20 and Sum<100 is true. The sum is evaluated inside the loop.

Another feature is that any of the section of the **for** loop can be omitted, if necessary.

Program:-

```
# include<stdio.h>
main()
{
    int x;
    x = 1;
    for(; x<20;)
    {
        printf ("%d \n", x);
        x=x+2;
    }
}
```

Both the initialization and the increment section of the for loop hare been omitted. The initialization is done before the for statement and the control variable is incremented inside the for loop. In such cases, the sections are left empty but the semicolons must remain there. If all the sections of the **for** statement are left empty, it forms an infinite loop and the loop continues to execute and the only way to terminate the loop is to use the **jump statements**.

Program:-

```
# include<stdio.h>
main()
{
    char ch = ' \o';
    for (; ;)                    /* execute forever */
{
    ch = getchar();
    if(ch == 'A')
        break;
}
    printf("You typed A);
}
```

This loop will run until A is typed at the keyboard.

for loop can also be executed without any loop body. This is useful to set up time delays for example:-

```
for(x=1;x<200;x++)
```

The semicolon after the for statement implies that it is a null statement and hence no error is reported.

2.10 While loop:-

Syntax:-

```
While (condition)
{
    statements;
}
```

The while is an entry controlled loop statement. The condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the condition is again evaluated and if it is true, the the body is once again executed. This process is repeated until condition finally fails, at which point the control is transferred out of the loop.

Program:-

```
# include<stdio.h>
main()
{
    y=0;
    x=1;
    while(n<=20)
    {
        y= y+x;
        x=x+1;
    }
    printf("sum = %d \n", y);
}
```

The body of the loop is executed 20 times, each time adding the number x, which is incremented inside the loop.

Program:-

```
# include<stdio.h>
main()
{
    char ch;
    ch = ' \o ';
    while (ch ! = 'A')
    {
        ch=getchar();
    }
}
```

First, the character ch is intialised to ' \ o'. The while statement then begins by testing whether ch is not equal to 'A'. Since in the first attempt, ch is assigned ' \o', the test is true and the loop body is executed. Each time a letter is keyed is, the test is carried out and loop statement executed until the letter 'A' is pressed. When 'A' is pressed, the condition becomes false, and the loop terminates.

Program:-

```
# include<stdio.h>
main()
{
    int count;
    float x,y;
    printf("Enter the value of x: ");
    scanf("%f", &x);
    y=1.0;
    count = 1;
    while(count<=4)
    {
        y = y * x;
        printf("y=%f \ n", y);
        count++;
    }
}
```

Output:–

```
Enter the value bof x : 2.5
    y = 2.50000
    y = 6.25000
    y = 15.625000
    y = 39.06250
```

The above program initializes y, and then multiplies by x, count number of times. where count = 1,2,3,4 For every count, the value of y*x is displayed. When the count exceeds 4, the loop terminates.

Do – while statement:–

Unlike the for and while loops that test the loop condition at the top of the loop, the do – while loop checks its condition at the bottom of the loop. This means that a **do – while** loop always executes atleast once.

Syntax:–

```
do
{
    statements;
} while (condition);
```

On reaching the do statements, the program proceeds to evaluate the body of the loop first. At the end of the loop, the condition in the while statement is evaluated. If the conditions is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and the control goes to the statement after the loop code.

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

The most common use of the do – while is in menu selection routine. When a valid response is typed, it is returned as the value of the function

Program:–

```
# include<stdio.h>
main()
{
    char ch;
    printf("1. Check spellings \n");
    printf("2. Correct spellings errors \ n");
    printf("3. Display selling Errors \ n");
    printf(" Enter your choice : \n");
    do
    {
        ch = getchar();
switch(ch)
```

```
    {
        case ' 1 ' : check_spelling();
                     break;
        case ' 2 ' : Correct_ errors();
        case '3' : display_errors();
                   break;
    }
} while (ch! = '1' && ch! = '2' && ch! /= '3' )
}
```

Jump statements:–

Loops perform a set of operations until the control variable fails to satisfy the condition. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs. C permits a jump from one statement to another within a loop as well as jump out of a loop. This can be accomplished by the following jump statements:

1. break
2. exit
3. continue

Break statement:–

The break statement has two uses. The first is to terminate a case in the switch statement. The second is to force the termination of a loop, by passing the normal loop conditional test.

The break statement at the end of each case signals the end of that particular case, and cause an exit from the switch statement transferring the control to the next line following the switch.

When break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

Program:–

```
# include<stdio.h>
main()
{
    int t,p;
    sum=0;
    for(t=0; t<100; t++)
    {
        sum=sum+t;
```

```
            printf(" %d \t %d \n",t,p);
            if (p>100) break;
        }
    }
```

If the break statement is used in nested loops, the break will cause an exit only from the inner most loop.

Example:–

```
    for(t=0; t<100; t++)
    {
        count = 1;
        for( ; ; )
        {
            printf("%d", count);
            count ++;
            f(count==10) break;
        }
    }
```

prints members, through 10 on the screen 100 times Each time the break is encountered, control passed back to the outer for loop.

A break used in a switch statement affects only that switch and not any loop the switch happens to be in.

Exit():–

The function exit() found in standard library causes an immediate termination of the entire program. The exit() function stops program execution and forces a return to the operating system. The exit() function has the general form

Void exit (int status);

Common use of exit() occurs when a mandatory condition for program's execution is not satisfied. For example, imagine a computer game for which a color graphics must be present in the system the main() of this game may look like this.

```
    # inciude<stdio.h>
    main()
    {
        if(!color_card())
        {
```

```
            exit 1;
        }
        play();
    }
```

where the color_card() is a user defined function that returns true if the card is present. If the card is not present, color_card() returns false and the program terminates.

2.11: Continue statement:-

The continue statement works some what like break statement. But instead of forcing termination, continue forces the next iteration of the loop to take place, skipping any code in between. for eg. the following routine displays only positive numbers.

```
do
    {
        scanf("%d",&x);
        if(x<0) continue;
        printf("%d", x);
    } while (x! = 100)
```

In while and do–while loops a continue statement causes control to go directly to the conditional test and then continue the looping process. In the case of for, first the increment part of the loop is performed, next the conditional test is executed, and finally the loop continues.

The above eg. can be changed to only allow 100 numbers to be printed.

```
for(t=0; t<100; t++);
{
    scanf("%d", t);
    if(x<0) continue;
    printf("%d",x);
}
```

In the following fragment code, continue is used to expedite the exit from the loop by forcing the conditional test to be performed sooner.

```
Char done, ch;
done = 0;
while (! Done)
{
```

```
        ch = getchar();
        if(ch== '$')
        {
            done = 1;
            continue;
        }
        putchar (ch+1);
    }
```

GOTO:–

Though C is a structured programming language, it contains the following unstructured forms of program control.

goto, Labels

A got statement transfers control to any other statement within the same functions in a C program, but it allows jumps in and out of blocks. Therefore it violates the rules of strictly structured programming language.

Syntax:–

2.12: Goto Label;

Where label is an identifier, which must appear as a prefix to another C statement in the same function. The semicolon after the label identifier marks the end of the goto statement.

Goto statement in a program makes it difficult to read. They reduce program reliability and make program difficult to maintain. They are however used because they can provide a useful means of existing from within deeply nested loops. Consider the following

```
for(. . )
{
    for(. .)
    {
        if(. .)
        goto error1;
        :
    }
}

error1: printf("Error ! ! ");
```

as seen, the label appears as a prefix to another statements in the form,

```
label: statement
or
label:
{
    statement sequence
}
```

Questions:

1. Write a program to find whether given number is odd or even.
2. Write a program to generate Fibonacci series using while loop.
3. Write a program to accept ten integers and to find the maximum and minimum integer.
4. Write a program to accept any members and spell it out.
5. Write a program to accept any number and to check whether it is an Armstrong member or not.
6. Write a program to calculate the following expression. X=(53 - 23) / 52 + 42.
7. Write a program to accept a number in the range 1 – 12 and spell.
8. The month which falls on that number for instance ⇒1 Jan,⇒2 Feb.
9. Write a program to find the sum of first 20 odd numbers using Goto and if statement.
10. Write a program to find the greatest among 3 numbers without using if else.
11. Write a program to find the sum of first 20 prime numbers.

UNIT – 3
FUNCTIONS

Introduction:

A function is a self – contained program segment that carries out a specific, well defined task. A large problem has to be split into smaller segments so that it can be efficiently be solved. This is where, functions come into the picture. They are actually the smaller segments, which help solve the larger problem.

Functions are the building blocks of C in which all program activities occur. In C, the functions have been categorized into user – defined functions and library or in–built functions. The major distinction between these two categories is that library functions are not required to be written by us whereas a user defined function has to be developed by the user at the time of writing a program.

The use of user – defined functions allows a large program to be broken down into a number of smaller, self contained components, each of which has some unique, identifiable purpose. Thus, a C program can be modularized through the intelligent use of functions.

Uses of modular approach:

Many programs require that a particular group of instructions be accessed repeatedly from different places within the programs. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed. Thus the use of a function avoids the need for repeated programming of the same instructions.

Functions provide logical clarity to the program. The decomposition of the program into several concise functions make the programs easier to understand, write and debug, and their logical structure is more apparent than programs which lack this type of structure.

The use of functions also enables a programmer to build a customized library of frequently used routines. Each routine can be programmed as a separate function and stored within a special library file. Hence a single function can be used by a number of different programs. This way, it promotes portability since programs can be written that are independent of system dependent features.

User defined functions in a program:

Every C program consists of one or more functions.

A function will carry out its intended task whenever it is accessed ("called") from some other portion of the program. The same function can be accessed from several different places within a program. Once the intended task of the function has been carried out, control returns to the point from which the function was called.

Syntax:

```
type specifier return – type func–name (parameter – list)
{
   local variable – declaration;
   statement sequences;
   return (expression);
}
```

Generally, a function will process information that is passed to it from the calling portion of the program, and return a single value of the data type given by the type specifier. Information is passed to the function via variables called parameters and returned via the return statement. Some functions however, accept information but do not return anything (eg. Printf) whereas other functions (eg. Scanf) return multiple values.

3.1 Defining a function:

A function definition has 2 principal components. The first line (including the argument declaration) and the body of the function.

The first line of a function definition contains the data type of the value returned by the functions, the function name followed by a set of arguments, separated by commas and enclosed in parentheses. Each argument is preceded by its associated type declaration. An empty pair of parentheses must follow the function name if the function definition does not include any arguments.

In general, the first line can be written as

Return–type func–name (type1 arg1, ... type n argn)

Where return – type represents the data type of the item that is returned by the function, func–name represents the function name, and type1, type2 . . . . type n represent the data types of the arguments arg1,arg2, ... argn. The return type is assumed to be type int if it is not explicitly given. The arguments are called formal parameters of the function.

In this function definition a and b are formal parameters.

The rest of the function definition is the compound statement that define the action to be taken by the function. This compound statement is sometimes referred to as the body of the function.

Example:

```
int mul (intx,inty)
{
   in t n;
   n = x * y;
   return n;
}
```

Here, the function name is mul, followed by the formal parameters x and y with their data types (here, int, in both cases). The return type is also int, this is the value that will be returned to the calling function. Within the braces is the body of the function mul. It actually carried out the multiplication of two numbers.

3.2 Calling function:

A function can be called by simply using the function name, followed by a list of arguments enclosed in parentheses and separated by commas. The arguments in the function call are called actual parameters, in contrast to the formal parameters that appear in the first line of the function definition. There will be one actual argument for each formal argument. The actual arguments may be expressed as constants, single variables, or more complex expressions. However, each actual parameter must be of the same data type as its corresponding formal argument.

Example:

```
Main ()
{
   int mul (intx, inty);
   P = mul (a,b);
   printf ("%d",P);
}
```

When the compiler encounters a function call the control is transferred to the function mul (x,y). This function is then executed line by line and a value is returned when a return statement is encountered. This value is assigned to P.

A function, which returns a value, can be used in expressions like any other variable.

Each of the following statements is valid;

```
printf ("%d", mul (p,q));
y = mul (p,q) / (p+q);
if (mul (p,q) > total) printf ("large");
```

However, a function cannot be used on the right side of an assignment statement.

For instance,

```
mul (p,q) = 50; (this is incorrect)
```

A function that does not return any value may not be used in expressions; but can be called to perform certain tasks specified in the function such functions may be called in simple by stating their names as independent statements.

```
# include < stdio.h >
main ()
{
   int x,y;
   void swap (int a, int b); / * function declaration returning no value */
   printf ("the value of x and y are:");
   Scanf ("%d%d", &x,&y);
   Swap (x,y)  / * function called as independent statement,
     with actual parameters  */

   printf ("The values of x and y after exchange are %d%d: x,y);
}
Void Swap (inta,intb)  /* function definition with formal parameters */
{
   int temp;
   temp = x;  / * S are the value of x in variable temp */
   x = y;     / * put y into x * /
   y = temp;  / * put x into y * /
}
```

3.3: Categories of Functions:

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

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

No Arguments and No Return values:

A function may not have any arguments, and it may not also return any value to its calling program. In effect, there is no data transfer between the calling function and the called function. A function that does not return any value cannot be used in expressions.

Program to illustrate the use of functions that do not take any arguments and do not return any value.

```
# include < stdio.h >
main ()
{
   printline ();
   printf ("User Defined Functions");
   printline ();
}
Void printline ()
{
    int n;
    for (n=1; n < 40; n++)
    {
      printf ("–");
    }
    printf ("\n");
}
```

The above program has two user defined functions; main () and printline () main () is the calling function that calls printline() function. The printline () function has no arguments, so there are no argument declarations. The printline () when encountered, prints a line of 39 character length. After executing the printline () function, the control is again transferred back to the main. Since everything is done by printline () itself there is in fact nothing left to be sent back to the calling function.

Note that there is no return statement employed, the closing brace of the function signals the end of execution of the function, thus returning the control, back to the calling function.

Arguments with No Return values:

In the above program, the main () has no control over the way the function receive input data. For example, the function printline () will input a line each time it is called. We could make the calling function to read the data from the terminal

and pass it on to the calling function. We can modify the declaration of printline () to include arguments as follows:

```
Printline (ch)
```

The calling function can now send values to these arguments using function calls containing appropriate arguments.
Consider the modified program:

```
#include < stdio.h >
main ()
{
   printline ('A');
   printf ("user Define Function");
   printline ('x');
}
printline (int ch)
{
   int n;
   for (n = 1; n < 15; n++)
  {
    printf ("%c", ch);
   }
    printf ("\n");
}
```

Output:

The function printline() is called twice. The first call passed the character 'A' while the second passes the character 'X' to the function. These are assigned to the formal parameter ch for printing lines.

```
# include < stdio.h >
main ()
{
   float principal inrate;
   int period;
   printf ("Enter principal amount, interest");
   printf ("rate, and period \n");
   Scanf ("%f%f%d", & principal, & in rate, & period);
   Printline ('Z');
   Value (principal, in rate, period);
   Printline ('C');
}
```

```
printline (ch)
   char ch;
{
    int i;
    for (i = 1; i < = 52; i ++)
    {
        printf ("%C",ch);
        printf ("\n");
    }
}
value (p,r,n)
   in t n;
   float p,r;
{
   int year;
   float sum;
   sum = P;
   year = 1;
   while (year < = n)
   {
        Sum = sum * (1+r);
        Year = year + 1;
   }
        printf ("%f\t%f\t%d\t%f\n") p,r,n,sum);
}
```

Output:

```
Enter principal amount, Interest rate and period
5000        0.12                5
5000        000000      0.120000     5      88.11.708984
```

Arguments with Return values:

Still another variation can be that a function cannot only have arguments but can also return a value to the calling function.

Program to calculate the factorial of a number:

```
# include < stdio.h >
main ()
{
   in t num;
   long int fact;
   long int factorial (in t n);
   printf ("Enter the number:");
```

```
   Scanf ("%d", & num);
  fact = factorial (num);  / * num as an actual parameter*/
  printf ("%ld", fact);
}
long int factorial (int n) /* n as a formal parameter */
  {
     int ;
     long int prod = 1;
     if (n,1)
         for (i = 2; i < n; i ++);
           prod * ; = i;
  }
```

Here the function definition, factorial () takes int n as a formal parameter which takes its value from the actual parameter passed to the function factorial () in the calling function, main (). After executing factorial (`), a value prod is returned to the variable fact in the calling function main (). This is the value that is printed on the screen by the printf statement.

Program:

```
# include < stdio.h >
main ()
{
   float principal, rate, amount;
   int time
   float value (float p, float r, int t);
   printf ("Enter the principal, interest rate, period:");
   Scanf ("%f%f%d", & principal, & rate, & time);
   amount = value (principal, rate, time);
   printf (%f\t%f\t%d\t%f\n", principal, rate, time, amount);
}
float value (float p, float r, int t);
{
   int year;
   float sum;
   sum = p;
   year = 1;
   while (year < = n)
     {
        sum = sum + (1 +r)
        year = year +1;
     }
        return (sum);
}
```

The above program calculates total amount after adding the simple interest earned on the principal amount. The calculated value is passed on to main () through the statement;

Return (sum);

The value of sum is assigned to the variable amount by the function call:

amount = value (principal, rate, time);

which in turn is printed on the screen by the printf () statement.

Program to convert lowercase character to upper case using user-defined function:

```
# include < stdio.h >
main ()
{
   char cl,lower,upper;
   char to _ upper (char cl);
   printf ("Enter a lowercase character:");
   Scanf ("% c", & lower);
   upper = to . upper (lower);
   printf ("The uppercase equivalent of % c is % C", lower,upper);
}
   char to . upper (char cl)
     {
       char c2;
       C2 = (c1 > = `a' & & c1 < = `¬')? (`A' + c1 - `a'): c1;
       return (c2);
     }
```

This program consists of two functions main and to upper.

The main function contains the declaration of the function to _ upper, like any other variable. The function has to be explicitly declared with all its parameters and return type before actually accessing it. When the function is defined, it is defined with the same parameters and the return type.

Function main reads the character and assigns it to the char type lower. Main then calls. Note that the function to upper carries out the actual conversion. A lower case letter is transferred into the function via an argument c1, and the upper case equipment, c2; is returned to the calling portion of the program (ie.) main via the return statement.

3.4 Passing Arguments to a function:

In general, the functions can be passed arguments in one of two ways. The first is called call by value. This method copies the value of an argument into the formal parameter of the function. Therefore, any changes that one made to the formal parameters do not affect the actual parameters.

Example:

```
# include < stdio.h >
main ()
{
   int sq r (int a);
   int t = 10
   printf ("%d%d",t, sqr (t));
}
   sqr (int a)
   {
       a = a * a;
       return  (a)
   }
```

In this example, the value of the argument to sqr (), 10 is copied to the parameter a When the assignment a = a * a takes place, the only thing modified is the value of a. The value of t, used to call sqr () still has the same value is 10. The value of a is finally returned to the main function.

Output:

```
10          100
```

Remember that only a copy of the value of the argument is passed to the function What occurs inside the function has no effect on the variable used in the call

Call by reference is another way in which the argument can be passed to a function. In this method, the address of an argument is copied to the formal parameter. Inside the function, the address is used to access the actual parameter used in the call. This means that changes made to the formal parameter affect the variable used to call the function.

Example:

```
# include < stdio.h >
main ()
{
   int x, y;
```

```
    void swap (int *x, int * y);
    printf ("The values of x and y are:");
    Scanf ("%d%d", & x, & y);
    Swap (&x, &y)     /* addresses of variables x and y passed to the
                          function  */
    printf ("The values of x and y after exchange are %d %d:", x,y);
}
   void swap (int *x, int y)   /* pointing to the addresses of x and y */
   {
        int temp;
        temp = * x; /* saring the value at address x, in variable temp */
        *x = *y;        /* put y in to  x */
        *y = temp;      /* put x into y */
   }
```

In the above example, the function swap () is called with the addresses of the arguments using the operator.

While executing the function Swap (), * operator is used to access the values, present at the addresses of x and y. Therefore, it is the same values used to call the function that are interchanged.

Nested functions:

C permits resting of functions freely, main can call function 1, which calls function 2, which calls function 3 and so on: There is in principle no limit as to how deeply functions can be nested.

Program:

```
main ()
{
   int a,b,c;
   float ratio (int x, int y, int z);
   Scanf ("%d%d%d", &a,&b&c);
   printf ("%f\n", ratio (a,b,c);
}
float ratio (int x, int y, int z)
  {
        : f (difference (y,z))
              return (x/(y–z));
        else
              return (0,0);
  }
   difference (int p, int q)
```

```
{
   :f (p; = q)
      return (1);
   else
       return (0);
}
```

The above program calculates the ratio a/(b–c) and prints the result. We have the 3 following functions.
Main (), ratio (), difference ()

Main () reads the values of a,b,c calls function ratio () to calculate the value a/(b–c). This ratio cannot be calculated if (b–c) = 0. Therefore, ratio () calls another function. difference () to test whether the difference b–c is 0 or not difference () returns 1, if b is not equal to C. Otherwise returns 0 to the function ratio (). In turn, ratio calculates the value a / (b–c) if it receives value 1 and returns the result in float. In case, ratio receives zero from difference, it sends back 0. 0 to main indicating that (b – c) = 0

Nesting of function calls is also possible.
P = mul (mul (5,5),3); is valid.

3.5. Recursion:

In C, functions can call themselves. A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself.

A simple example is the function fact () which computes the factorial of an integer. The factorial of a number N is the product of all the whole numbers from 1 to N.

Program:

```
# include < stdio.h >
main ()
{
   int num;
   long int fact;
   long int factorial (int n);
   printf ("Enter the number:");
   Scanf (%d", & num);
   fact = factorial (num);
   printf ("%ld", fact);
}
long int factorial (int n);
   {
        int ans;
   if (n = = 1) return 1;
```

```
    ans = factorial (n - 1) * n;
    return (ans);
}
```

When factorial () is called with an argument of 1 the function returns 1, otherwise it returns the product of factorial (n – 1) * n. To evaluate this expression, factorial (n–1) *n. To evaluate this expression, factorial () is called with n-1.. This happens until n equals 1 and the calls to the function begin returning.

Computing the factorial of 3, the first call to factorial () causes the second call to the function to be made with the argument of 2. This causes the value of ans to be 2 * 3. Again the function is called with an argument of 1. This returns value 1, which is multiplied by the value of ans. The ans is then 1 * 2 * 3 or 6.

When a function calls itself , new variables and parameters are allocated storage on the stack, and the function code is executed with these new variables from the beginning. A recursive call does not make a new copy of the function. Only the arguments are new. As each recursive call returns, the old variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

Program:

```
main ()
{
    int a,b;
    a = sumdig (123);
    b = sumdig (123);
    printf ("%d%d", a,b);
}
    sumdig (int n)
    {
        static int s = 0;
        in t d l;
        :F (n ! = 0)
        {
            d = n % 10;
            n = (n - d) / 10;
            S = S + d;
             Sumdig (n);
        }
            else
                return (S);
}
```

Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack space could become exhausted. If this happens, a stack overflow occurs.

Consider the following example:

```
main ()
{
   printf ("Blast It");
   main ()
}
```

Hence, the main () function calls itself, prints Blast It, again calls itself, again prints the message. This process goes on till the stack overflows.

3.6. Macros and Preprocessors:

Preprocessor is one of the unique features of C language. It is used to make the program easy to read, easy to modify, portable, and more efficient. The preprocessor, as its name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as preprocessor directives are placed in the source program before the main line.

Preprocessor directives follow special syntax rules, they all begin with the symbol # in column one and do not require a semi colon at the end.

A set of commonly used preprocess directives and their functions are given below.

Directive	Function
# define	defines a macro substitution
# undef	undefines a macro
# include	specifies the files to be included
# ifdef	tests for a macro definition.
# endif	specifies the end of # if
# ifndef	test whether a macro is not defined
# if	tests and compile time condition
# else	specifies alternatives when # if test fails

These directives can be divided into 3 categories;

1. Macros substitution directives
2. File inclusion directives
3. Compiler control directives

Macro substitution:

Macro substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. The preprocessor accomplishes this task under the direction of # define statement. This statement, usually known as a macro definition takes the following general form.

```
# define identifier string
```

The string may be any text, while the identifier must be a valid C name

Single macro substitution:

Single string replacement is commonly used to define constants.

Eg:

```
# define    COUNT    00
# define    FALSE     0
# define      PI      3.1415926
```

It is a convention to write all macros in capitals to identify them as symbolic constants.

Macro with arguments:

The preprocessor permits us to define more complex and more useful form of replacement. It takes the form

 # define identifier (f1, f2, ... fn) string

There is no space between the macro identifier and the left parentheses. The identifiers f1, f2, ... fn are the formal macro arguments that are analogous to the formal arguments in a function definition. Some commonly used definitions are;

```
# define    MAX (a,b)    ((a) > (b)) ?  (a) : (b))
# define    MIN (a,b)    ((a) > (b)) ?  (a) : (b))
# define    ABS (x)      ((x) > (0)) (x)  : (– (x))?
# define    STREQ (s1,s2) (strcmp (s1), (s2) ==0)
```

Nesting of macros:

We can also use one macro in the definition of another macro (ie) macro definition may be nested. For instance, consider the following macro definitions.

```
# define    M            S
# define    SQUARE (x)   ((x) * (x))
# define    CUBE (x)     (SQUARE (x) * (x))
```

Undefining a macro:

A defined macro can be undefined, using the statement # undef identifier. This is useful when we want to restrict the definition only to a particular part of the program.

Questions:

1. Write a program to accept the number and to change the sign.
2. Write a program to find the factorial using functions.
3. Write a program using functions to accept an amount and tender changed with maximum number of currency notes.

   For instance, 468

   100 * 4, 50 * 1, 10 * 1, 5 * 1, 2 * 1, 1 *1.
4. Accept a string and change the case.
5. Write a program to find the vowels, consonants, special symbols in a given string.
6. Using function calculate depreciation.

3.7. Library functions:

The C language is accompanied by a number of library functions that carry out various commonly used operations or calculations.

A library function is accessed simply by writing the function name, followed by a list of arguments that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The arguments can be constants, variable names, or more complex expressions. The parentheses must be present, even if there are no arguments.

Function	Type	Purpose
abs (i)	int	Return the absolute value of;
Ceil (d)	double	Round up to the next integer value
Cos (d)	double	Return the cosine of d
Cosh (d)	double	Return the hyperbolic cosine of d
exp (d)	double	Raise e to the power d
fabs (d)	double	Return the absolute value of d
floor (d)	double	Round down to the next integer value
fmod (d1,d2)	double	Return the remainder of d1|d2 with same sign as d1.

get char ()	int	Enter a character from the standard input device.
log (d)	double	Return the natural logarithm of d.
pow (d1,d2)	double	Return d1 raised to the d2 power
printf ()	int	send data items to the standard output device
putchar (c)	int	send a character to the standard output device
rand ()	int	Return a random positive integer
sin (d)	double	Return the sine of d
sqrt (d)	double	Return the square root of d
srand (u)	void	Initialise the random number generator
Scanf ()	int	Enter data items from the standard input device
tan (d)	double	Return the tangent of d
toascii (c)	int	convert value of argument to ASCII
tolower (c)	int	convert letter to lower case.
toupper (c)	int	convert letter to upper case.

UNIT – 4
STORAGE CLASSES

Introduction:

Every C variable has a characteristic called its storage class. The storage class defines two characteristics of the variable, its storage class. The storage class defines two characteristics of the variable; its lifetime and its scope. The lifetime of a variable is the length of time it retains a particular value. The scope of a variable refers to which it retains a particular value. The scope of a variable refers to which parts of a program will be able to function, a file, a group of files, or an entire program.

From the C compilers point of view, a variable name identifies some physical location within the computer, where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept memory or one of the CPU registers. The variable's storage class determines where it should be stored in memory or in a register.

There are 4 storage classes in C these are

1. Automatic
2. Global
3. Static
4. register

Syntax:-

Storage_specifier data_type var_name;

4.1: Automatic variables:

These are nothing but the local variables. The scope of an automatic variable can be smaller than the entire function. If is declared from within a compound statement. The scope is then restricted to that compound statement. The scope is then restricted to that compound statement. They can be declared using the specifi‑r auto, though the declaration is not necessary. Any variable declared within a function or a code block are automatic assumed to be of the class auto and the system. It aside the required memory area for them.

Program to illustrate automatic variables:-

```
#include<stdio.h>
main()
{
   int a=10, b=20;
    printf("In main function the value of a is %d and the value of b is % d, a, b);
    func1 ();
     printf("In main function again the value of a is %d and the value of be
               is %d, a,b);
     printf("\nEnd of main function);
}
func1()
{
   int a= 20, b=30;
   printf("\n In func1, the value of a is %d and the value of b is %d ", a,b);
   func2();
    printf("\nIn func1 again, the value of a is %d and the value of b is %d",a,b);
   printf("\n End of func1");
}
func2()
{
    int a=20,b=30;
    printf("\n In func2, the value  of a is %d and the value of b is %d",a,b);
    func2();
    printf("\n In func2 again the value of a is %d and the value of bis %d", a,b);
    printf("\n End of func2");
}
```

Out put:-

```
In main function, the value of a is 10 and the value of b is 20
In main func1, the value of a is 30 and the value of b is 40
In main func2, the value of a is 50 and the value of b is 60
End of func2
In main func1, the value of a is 30 and the value of b is 40
End of func1
In main func1, the value of a is 10 and the value of b is 20
End of main function.
```

4.2: External variables:-

Global variables:-

Variables that are both alive and active through out the entire programs are known as external variables. They are also known as global variables unlike local variables, global variables can be accessed by any function in the program External variables are declared outside a function. A program to illustrate the properties of global variable is presented below. Note that variable x is used in all functions, but more except fun2, has a definition for x. Because x is used in all functions it is available to each function without having to pass x as a function argument. Once a variable was been declared as global, any function can use it and change its value. Then subsequent functions can reference only that new value.

```
#include<stdio.h>
main()
{
   x=10;
   printf("x = %d\n",x);
   printf("x = %d\n",fun1() );
   printf("x = %d\n",fun2() );
   printf("x = %d\n",fun3() );
}
fun1()
{
   x=x+10;
   return(x);
}
fun2()
{
   int x;
   x=1;
   return(x);
}
fun3()
{
    x=x+10;
    return(x);
}
```

Output:-

```
x=10
x=20
x=1
x=30
```

program:-

```
#include<stdio.h>
int a,b;
main()
{
   a=10,b=20;
   printf("\n In main function the value of a is %d and value of b is %d",a,b);
   func1()
   printf("\n In main function again, the value of a is %d and the value of b
            is %d", a, b);
   printf("\n End of main function");
}
func1();
{
a=30,b=40;
printf("\n In func1 again, the value of a is %d and the value of b is %d"
      a,b);
func2();
printf("\n In func1 again, the value of a is %d and the value of b is %d"
      a,b);
printf("\n end of func1");
}
func2()
{
a=50,b=60;
printf("\n Infunc2,the value of a is %d and the value of b is %d", a,b);
printf("\n End of func2");
}
```

Out put:-

```
In main function, the value of a is 10 and the value of b is 20
In main func1, the value of a is 30 and the value of b is 40
In main func2, the value of a is 50 and the value of b is 60
End of func2
In main func1, the value of a is 50 and the value of bis 60
End of func1
In main func1, the value of a is 50 and the value of b is 60
End of main function.
```

4.3: Static variables:-

Static variables that are available in a function in which they are defined and they do not loose their value when the function is terminated. This type of variables are present throughout the program but are only accessible in the function in which they were originally defined. To define static variables, the static reserved word is used.

A static variable may either be an internal static variable, depending on the location of declaration.

Internal static variables are the variables that are defined within a function. It can be only used in the function in which they are declared. Internal static variables can be compared with the automatic variables with respect to the scope of the variables which is the function in which these are defined.

When we use the static modifier, we cause the contents of a local variable to be preserved between calls. Also, unlike normal local variables, which are intialised each time a function is entered, a static local variable is intialised only once.

Program:-

```
#include<stdio.h>
main()
{
   int I;
   for(I=0;I<5;I++)
   rad();
}
   rad()
   {
   static int s=0;
   s++;
   printf("\ns=%d",s);
   }
```

output:-

```
s = 1
s = 2
s = 3
s = 4
s = 5
```

In the above program, the first call to the function rad, declares the variables S as static integer and intialises it to o. Subsequent calls increase the value stored in the variable S.

External static variables or global static variables differ the internal static variable in the manner that internal static variables are available only to the function in the program, file in which the variable is defined and not in other program files where as external static variables are available across program files.

4.4: Register variables:-

In a computer, the variables can be either be stored in the memory or in the registers variables that are to be stored in the registers are declared using the register keyword before the variable declaration.

Registers are similar to memory locations with the difference that they are a part of the processor itself therefore operations on register are faster than those relating to memory.

Usually, the variables that are frequently used in a program whose access speed is important are assigned to register storage class.

A variable declaration with register storage class is a request to the compiler that the variable I to be assigned to a register. The declaration is a request because of the limited member of registers available on the machine. The compiler tries to accommodate as many variables as possible to registers, but it may not be able to accommodate all of them due to non-availability of registers. The compiler will automatically convert register variables to non-register variables once all the storage space on the registers is filled up only local variables can be declared to belong to the register storage class.

Because a register variable may be stored in a register of the CPU, it may not have a memory address. This means that we cannot use the '&' to find the address of a register variable.

Program:-

```
#include<stidio.h>
main()
{
   register int i;
   int n,s,d;
   printf("\n the Armstrong number between 1 and 999");
```

```
 for(i=1;i<999;i++)
{
s=0;
n=i;
while(n)
{
d=n%10;
n=n/10;
s=s+(d *d*d);
}
if(s==i)
printf("\n%d",i);
}
}
```

Output:

The Armstrong number between 1 and 999

```
 1
152
370
371
407
```

In the above program, the value of i is varied from 1 to 999. For each of these values of i, the cubes of individual digits comprising i are added and the resultant S is tested against i. If these two values are equal i is displayed. Since i is used to control the looping, it is declared to be of the storage class register. This declaration increases the efficiency of the program.

4.5: Functions in Multiple programs:-

A file is composed of information stored as a separate entity within the computer or on an auxiliary storage device. A file can be a collection of data, a source program, a portion of a source program, an object program etc.

Programs can be composed of multiple files such program could make use of lengthy functions where each function may occupy a separate file. As with variables in Multifile programs, functions can also be defined as static or external. The scope of the external function is through all files of the program and it is the default storage class for functions. Static functions are recognised only within the program file and their scope does not extend outside it.

The function header will look like:
static fn_type fn_name (argument list)]
Or
extern fn_type fn_name (argument list)

The keyword extern is optional as it is the default storage class.

Function declarations for extern functions:-

Function declarations for class extern are nested when the functions defined in one file are accessed by another. The declarations are placed at the beginning of the file, before the first function definition. It is good practice to begin a function declaration with the storage class extern, even though it is not necessary. The scope of the functions is within the files in which it is either defined or declared. Function declaration can be written in the same way as function definition headers. Consider two files file1 and file2. The first file calls a function cal()1 which is defined in the second file. The definitions and declarations for the above will be as followed.

File1

```
#include<stdio.h>
extern void cal(void);
main()
{
   :
   cal();
}
```

File2

```
Void cal()
{
   :
   return ;
}
```

File 1 and File 2 are compiled separately and then liked together. The generated object program normally has the same name as the first file listed in the link command (ie) file.exe.hence the program can be executed under the name file.

Program :-

Simulation of a Game of chance

First file:-

```
#include<stdio.h>
#include <stdio.h>
#include<ctype.h>
#define SEED 12345
extern void play(void);
main()
{
    char answer = 'y'
    printf("welcome to game of CRAPS\n\n");
    printf("To throw the dice 1press RETURN\n\n");
    Srand(SEED); \*intialise the random number generator*/
    While (to upper (answer) ! = 'N')
        {
        play();
        printf("\n Do you want to play again?(Y/N)");
        scanf("%c",&answer);
         printf("\n");
         }
    printf("Bye, have a nice day");
}
```

Second file:-

```
#include<stdio.h>
#include<stdio.h>
static int throw (void);
extern void play (void);
{
int score1 , score2;
char dummy;
printf("\n please throw the dice....") ;
scanf("%c",&dummy);
printf("\n");
score1 = throw();
printf("\n%2d",score1);
switch(score1)
{
case7:          /*win on first throw*\
```

```
case 11:
        printf("congratulations!You  you win on the first throw
        break;
  case 2:                    /*lose on first throw*\
  case 3:
case 12:
          printf("Sorry you lose the first throw\n");

          break;
 case 4:         /*additional throws are required */
 case 5:
 case 6:
 case 8:
 case 9:
case10:
        do
          {
          printf("Throw the dice again.....");
          scanf ("%c",&dummy);
          score 2 = throw();
          printf("\n%2d",score 2);
          }
          while (score2! = score1 && score2! = 7);
        if(score2==score1)
           printf("you win by matching your first score \n");
        else
           printf("you lose by failing to match your first score \n");
           break;
        }
          return;
        }
static int throw(void)
        {
           float x,x2;
           int n1,n2;
x1 = rand()/32768.0;
x2 = rand()/32768.0;
n1 = 1+(int) (6+x1); \*stimulate first due */
n2 = 1+(int) (6**2); \*stimulate second date*/
return(n1+n2); /*score is sum of two dice*/
}
```

UNIT – 5
ARRAYS

Introduction:

An array is a group of related data items that share a common name. Many applications require the processing of multiple data items that have common characteristics. In such situations it is convienient to place the items into an array, where they will share the same name.The individual data items can be characters, floating–point numbers, etc.

Each array element is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets. The number of subscript determines the number of dimensionality of the array. For example name[i] refers to the element in one – dimensional array **name**. Similarly, students[i][j] refers to an element in the two dimensional array **students**.

5.1:Defining an array:

Arrays are defined in much the same manner as the ordinary variables, except that each array name must be accompanied by a size is specified by a positive integer expression, enclosed in square brackets. The expression, enclosed in square brackets. The expression is usually written as a positive integer constant.

One dimensional array syntax:

Storage – class data – type array[expression]; where storage – class refers to the storage class of the array, datatype is the datatype, array is the array name, and expression which indicates the number of array elements. The storage – class is optional default values are automatic for arrays that are defined within a function or a block and external for arrays that are defined outside the function

Eg:- int x[100];
    Char text[80];
    Static char message[25];
    Static float n[12];

In C, there is no bounds checking on arrays you could overwrite either end of an array and write into some other variable's data, or even into a piece of the program's code. It is the programmers job to provide bounds checking when it is needed.

It is sometimes convienient to define an array size in terms of a symbolic constant rather than a fixed integer quantity. This makes it easier to modify a

program that utilizes an array size can be altered simply by changing the value of the symbolic constant.

Program:- Lower case to upper case Text conversion:

```
# include<stdio.h>
# include<ctype.h>
# define SIZE80
main()
{
    char letter[SIZE];
    int count;
    for(count = 0; count<SIZE ", ++ Count)
        letter[count]=getchar();
    for(count=0; count<SIZE; ++ count)
        putchar(toupper(letter[count]));
}
```

Notice that the symbolic constant SIZE is assigned a value of 80. This symbolic constant, rather than its value appears in the array definition and in the two for statement therefore, in order to alter the program to accommodate a different size array, only the # define statement must be changed.

For eg. to alter the above program so that it will process a 60 – element array, the original # define statement is simply replced by #define SIZE 60

This one changes accommodates all of the necessary program alterations; there is no possibility that some required program modification will be overlooked.

Automatic array, unlike automatic variable, cannot be intialised. However external and static array definitions can include the assignment of initial values if desired. The intial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas.

Initialisation of Arrays:

Syntax:

```
Static type array _ name[size] = {list of values};
```

eg:

```
static int number [3] = { 0, 0, 0}
```

will declare the variable number as an array of size 3 and will assign 0 to each element . If the number of values in the list is less that the number of elements,

then only that many elements will be intialised. The remaining elements will be set to 0 automatically.

For eg:

Static float total[5] = {0.0, 15.75, –10};

Will intialise the first 3 elements to 0.0, 15.75, –10.0 and the remaining two elements to 0.

The size may be omitted. In such cases, the compiler allocates enough space for all intialised elements.

For eg:

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

Will declare the counter array to contain 4 elements with initial values 1. This approach works fine as long as we intialise every element of the array. Characters may also be intialised in the same manner.

eg: Static char name [ ] : {'W', 'A', 'L', 'L', 'I', 'S' } declares the name to be an array of 6 characters intialised with the string " WALLIS".

String and character array:

A string is an array of characters. A string in 'C' is defined as any group of characters enclosed between double quotation. Double quotation marks do not from part of the character string.

A string in 'C' is stored internally in character arrays. Since the string can be of any length, the and of the string is marked with the single character '\o', the null character, which has the ascii value zero.

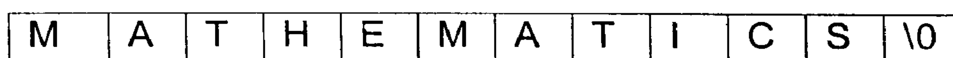
The string arrays are declared in 'C' in a similar manner as we had declared any other numeric array.

Syntax:

<char> <array.name> [<size>]

Suppose we want to store the string "MATHEMATICS" for this purpose, we need to define a character array of 12 elements.

The string will be stored in the character array as shown below.



null character indicating end of string

The fact that the string must be terminated by a null means that we must define the array that is going to hold a string to be one byte larger than the largest string it will be required to hold, to make room for the null.

The maximum size required to accommodate the above string into an array is 12 and not 11 as storage space for the null character at the end of every string has to be provided in the string declaration.

Intialisation of strings:

Strings can be intialised at the time of declaration or using a for loop in the program. At the time of declaration, the string can be intialised in any of the following manners.

1. char str[12] = “MATHEMATICS”;
2. char str[12] = {‘M’, ‘A’, ‘T’, ‘H’, ‘E’, ‘M’, ‘A’, ‘T’, ‘I’, ‘C’, ‘S’);
3. char str [ ] = “MATHEMATICS”;
4. char str[ ] = {‘M’, ‘A’, ‘T’, ‘H’, ‘E’, ‘M’, ‘A’, ‘T’, ‘I’, ‘C’, ‘S’, ‘\O’);

5.2 Passing an array to a Function:

An array can be passed to a function as an argument in a manner Similar to that used for passing variables. To pass an array to a called function, it is sufficient to list the name of the array, without any subscripts and the size of the array as arguments.

An entire array can be passed to a function as an argument the manner in which the array is passed differs marked ly, however, from that of the ordinary variable. When passing a one–dimensional array to a function, the function is called with the array name without any index. This passes the address of the first element of the array to the function.

Syntax:

```
<Storage_class> <type – specifier> <function – name>
                                        (<array–name>, <array –size>)
```

Program to pass array as an argument to a function

```
#include<stdio.h>
main()
{
    int array [5], I;
    printf("\n this program illustrate the use of
            arrays as parameters to a function");
    for(I=0; I<5; I++)
```

```
{
    printf("\n enter value for the element [%d]:" I+1);
    scanf("%d", & array[I];
}
    func1(array,5);
}
func1(array, 5);
}
func1(int arr[ ], intn)
{
    int z;
    for(z=0; z<5; z++)
    printf("\n the value of element [%d] is: %d", z+1, arr[z]);
}
```

When we pass arrays as arguments to a function we should remember one major distinction from passing ordinary variables and that is, if a function changes the value of an array elements, then these changes will be made to the original array that was passed to the function. This is in contrast to passing ordinary variables where the changes made in the called functions is not reflected in the calling function.

When the entire array is passed as an argument, the contents of the array are not copied into the formal parameter array, instead information about the memory addresses of the array elements are truly reflected in the orginal array in the calling function.

But, if we pass only one element of the array to a function, then this element is treated as ordinary variable, and a copy of it is made and passed to the array. Any changes to the array element passed as variable in the called function, is not reflected back in the calling function.

Program to illustrate that change in array element is reflected in the calling function:

```
# include<stdio.h>
main()
{
    int arr[5],z;
    func1(arr, 5);
    for(z=0; z<5;z++)
    printf("\n The value of element [%d] is : %d", z+1, arr[z]);
}
```

```
func1(int array[ ], int n)
{
    int l;
        for(l=0; l<5; l++)
        {
        printf("\n Enter value for the element [%d]: " i+1);
        scanf("%d",& array[l]);
        }
}
```

5.3:Two dimensional arrays:

C allows multidimensional arrays. The simplest form of multidimensional arrays is the two dimensional array. A two dimensional arrays are declared using this general form

type array – name [2nd dimension] [1st dimension]

Hence, to declare a two dimensional integer array d of size 10,20, you would write

```
int d[10] [20];
```

In the following example, a two dimensional array is loaded with the numbers 1 through 12, which it then displays on the screen.

```
# include<stdio.h>
main()
{
    int t,I,num [3] [4];
    for(t=0; t<3; ++t)
    {
        {
                num [t] [i] = (t*4) + i+1;
        }
    }
    for(t=0;t<3;++t)
    {
        for (i=0; i<4; i++)
        {
            printf("%d", num [t] [i]);
            printf("\n");
        }
    }
```

In this example, num[0][0] has the value 1; num [0][1], the value 2; num [0][2], the value 3; and so on. The value of num [2][3] is 12. Two dimensional arrays are stored in a row column matrix, where the first index indicates the row and the second indicates the column. This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order they are actually stored in the memory. In essence, the leftmost index can be thought of as a " pointer" to the correct row. The number of bytes of memory required by a two dimensional array is computed using the following formula

bytes = 2nd dimension * 1st dimension * size of(base – type);

Therefore, assuming 2 byte integers, an integer array with dimension 10,5 would have 10 × 5 × 2 or 100 bytes allocated.

When a two dimensional array is used as an argument to a function, only a pointer is passed to the first element. However, a function receiving a two – dim array as a parameter must minimally define the length of first dimension, because the compiler needs to know the length of each row if it is to index the array correctly. For example, a function that will receive a two–dim array with dimensions 5, 10 would be declared like this.

```
func (int x [ ] [10])
   {
      :
   }
```

You can specify the second dimension as–well but it is not necessary. The compiler needs to know the first dimension in order to work on statements such as

```
X [2] [4]
```

inside a function. If the length of the rows is not known, it is impossible to know where the next row begins.

The short program illustrates the use of two dim array to store the numeric grade for each student in a teacher's class. The program assumes that the teacher has 3 classes and a maximum of 30 students per class. Notice how the array grade is accessed by each of the functions.

```
# include<stdio.h>
# include<ctype.h>
# include<conio.h>
# include<stdlib.h>
# define CLASSES 3
```

```
    # define GRADES 30
    int grade [CLASSES] [GRADES];
}

Void disp _ grades (int g [ ] [GRADES]), enter _ grades (void);
int get _ grade (int num);
main()
{
    char ch;
    for(; ;)
        {
        do
            {
                printf(" (E)nter grades \n");
                printf("(R)eport grades \n");
                printf("(Q)uit grades \n");
                ch= to upper (getche ());
                } while (ch! = 'E' && ch! = 'R' && ch! = 'Q');

        switch (ch)
        {
        case 'E':
            enter _ grades ();
            break;
        case 'R':
            disp _ grades(grade);
            break;
        case 'Q':
            return (o);
        }
    }
}
void enter _ grades(void)
{
    int t,f;
    for(t=0; t<CLASSES; t++)
    {
        printf("class # %d: \n", t+1);
        for(f=0; f<GRADES; ++f)
            grade [t] [f] = get_grade(+);
    }
}
```

```
get_grade (int num)
{
    char s[80];
    printf("enter grade for student # %d: \n", num+1)
    get (s);
    return (atoi (s));
}
void disp–grades (int g [ ] [GRADES])
{
    int t,f;
    for(t=0; t<CLASSES;t++)
        {
            printf ("class # %d: \n",t+1))
            for(f=0;f<GRADES; ++f)
            printf("grade for student # %d is %d \n" f+1, g[t] [f];
        }
    }
```

Initialisation of two dimensional arrays:

Like one–dim arrays may be initialised by following their declaration with a list of intial values enclosed in braces.

For eg:

Static int table [2] [3] = {2,3,4,1,2,3};

Initializes the element of the first row to 2,3,4 and the second row to 1,2,3. The initialization is done by row. The above statement can be equivalently written as

Static int table [2] [3] = {{2,3,4}, {1,2,3}};

by surrounding the elements of each row by braces.

We can also intialise a two–dim array in the matrix form as shown below:

Static int table [2] [3] = {   {2,3,4}, {1,2,3} };

Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

If the values are missing in the intialiser. They are automatically set to zero. For instance the statement

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

Will initialize the first two elements of the first row to 1, the first element of the second row to 2, and all other elements to 0.

```
# include<stdio.h>
# define N 10
main()
{
    int i,j,n;
    float median, a[N],t;
    printf("Enter the number of items \n");
    scanf("%d", &n);
    printf("Input %d values \n", n);
    for(i=1; i<=n; ++i)
        scanf("%f", &a[i]);
    printf(" The sorted list is : \n");
    for(i=1; i<n; ++i)
    {
        for(j=1;j<n –I; ++j)
        {
            if (a[j] < = a[j+1])
            {
              t=a[j];
              a[j] = a[j+1];
               a[j+1]=t;
            }
            else
              continue;
        }
    }
    if(n%2==0)
        median = (a[n/2] + a[n/2+1]) / 2.0;
    else
        median = (a[n/2 + 1];
    for(i=1; i<=n; ++i)
        printf("%f",a[i]);
        printf("\n\n Median is %f \n", median);
}
```

Output:

Enter the number of items
5
Input 5 values
1.111 2.222 3.333 4.444 5.555
The sorted list is:
5.555000 .444000 3.333000 2.222000 1.111000
Median is 3.333000

Multi dimensional Arrays:

C allows arrays of more than two dimensions also. The general form of a multidimensional array is:

type array – name [size N] [size N–1] ...... [size 2] [size 1];

where size I is the size of the ith dimension.

Some examples are:

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

where survey is a three dim array declared to contain 180 integer type elements. Similarly, matrix is a four dimensional array containing 300 elements of floating point type.

Arrays of more than three dimensions are rarely used because of the amount of memory required to hold them.

When passing multi–dim arrays to a function, you must declare all but the leftmost dimension for example, if you declare array m as

```
int m[4] [3] [6] [5];
```

then a function, func(), receiving m, could look like

```
func (int d[ ] [3] [6] [5])
{
    :
    ;
}
```

of course, you are free to use the leftmost dimension if you like.

5.4: STRINGS:

In C, a string is defined to consist of a character array of any length that is terminated by a null. A null is specified as ' \o' and is 0. for this reason, it is

necessary to declare character arrays to be one character longer than the largest string that they are to hold.

For example:

Static char name [8] = { 'J', 'A', 'N', 'E', 'E', '\O' };

Similarly to declare an array s that holds a 10 – character string, you would write

Char s[11];

This makes room for the null at the end of the string.

C does not have string data types, but it still allows string constants. A string constant is a list of characters enclosed within double quotes. For example, here are two string constants.

"hello"
"good morning"

It is not necessary to add the null at the end of the string manually; the C compiler does this for you automatically.

Reading strings from Terminal:

Program to read a line of text from the terminal:

```
# include<stdio.h>
main()
{
    char line[80], character;
    int c=0;
    printf("Enter text, press <Enter> at the end \n");
do
    {
        character = getchar();
        line [c] = character;
        c++;
    } while (character! = ' \n' );
    c=c -1;
    line [c] = ' \o';
    printf("\n %s \n", line);
}
```

Output:

Enter text, press <Enter> at the end programming in c is interesting

Writing strings to screen:

Program to copy one string to another and copy the number of characters copied:

```
# include<stdio.h>
main()
{
    char str1[80], str2[80];
    int i;
    printf("Enter the first string \n");
    scanf("%s", str2);
    for(i=0;str2[i] ! = ' \o' ; i++)
    {
        str1[i] = str2[i];
    }
    str1[i] = ' \o';
    printf(" \n");
    printf(" The 2nd string is %s \n', str1);
    printf("Numbers of characters = %d \n",i);
}
```

Output:

Enter the first string
Very good
The 2nd string is
Very good
Number of characters = 10

We can also specify the precision with which the array is displaced. For instance, the specifications % 10 4 indicates that the first 4 characters are to be printed on a field width of 10 columns. However, if we include the minus sign in the specification (eg. %–10.4s), the string will be printed left justified. The following example illustrates the effect of various %s specifications.

```
# include<stdio.h>
main()
{
    static char city[15] = "united kingdom";
    printf("\n\n");
    printf("% 15s\n", city);
    printf("%5s\n",city);
```

```
    printf("%15.6s\n",city);
    printf("%-15.6 s\n", city);
    printf("% 15.0 s\n",city);
    printf("% 0.3 s\n",city);
    printf("% s\n", city);
}
```

Output:

```
United Kingdom
United Kingdom
         United
United
Un;
United kingdom
```

The printf on UNIX supports another nice feature that allows for variable field width or precision.

For instance,

```
printf("%* . *s", w,d, string);
```

prints the first d characters of the string in the field width of w, this feature comes in handy for printing a sequence of characters.

String – handling functions:

C library supports a large number of strings handling functions that can be used to carry out many of the string manipulations. Following are the most commonly used string – handling functions.

Function	Action
Strcat( )	Concatenates two strings
Strcmp( )	Compares two strings
Strcpy( )	Copies one string to another
Strlen( )	finds the length of the string

All these functions are in string.h headerfile.

Strcat( ) function:

The strcat( ) function joins two strings together.

Syntax:

```
strcat (string1, string2);
```

String 1 and String 2 are character arrays. When the function strcat( ) is executed, string 2 is appended to string 1. It does so by removing the null character from the end of the string 1 and placing the string 2 remains unchanged.

Program:

```
# include<stdio.h>
main( )
{
    char str1[20], str2[20], str3[20];
    printf(" \n enter a string \n");
    scanf("%s",str1);
    printf("\n The first string is %s", str1);
    printf("\n The second string is %s", str2);
}
```

Output:

```
Enter a string
Madurai

Madurai
Madurai
```

C permits nesting of strcat( ) functions. (ie) strcat (strcat(str2, str1), str3);

Is perfectly valid and conca tenates all the 3 strings. The resulting string is stored in str2.

Strcmp( ) function:

The strcmp( ) function compares two strings identified by the arguments.

Syntax:

```
Strcmp (string1, string2);
```

It returns 0 if the string are equal, greater than 0 if string 1 is greater than string 2 and less than 0 if string 1 is less than string 2. All comparisons are done lexicographically (according to dictionary order). String1 and string2 may be string variables or string constants.

eg:

```
strcmp(name1, name2);
strcmp(name1, "Rovina");
strcmp("Rovina", "Robin");
```

It is important to remember that the strcmp( ) function returns false if the strings are equal.

Strcpy( ) function:

Strcpy( ) function also accepts two strings as arguments.

Syntax:

Strcpy (string1, string2);

Its first argument is generally an identifier that represents the string. The second argument can be string constant or a string identifier. The function copies the value of string2 to string1. Hence, it effectively causes one string to be assigned to another string.

eg: strcpy(city, " Madurai");
will assign the string "Madurai" to the string variable city. The result will be the new value stored in string city. Similarly, the statement

strcpy(city1, city2);

will assign the contents of the string variable city 2 to the string variable city1. The size of the string array city1 should be large enough to receive the contents of city2.

Strlen( ) function:

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

var len (string);

where var is an integer variable which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character.

Program:

```
# include<stdio.h>
# include<string.h>
main( )
{
    char s1[80],s2[80];
```

```
    printf("enter the first string \n");
    scanf("%s", s1);
    printf("enter the second string \n");
    scanf("%s",s2);

    printf("The length of first string is %d and second string is %d",
                                                        strlen(s1), strlen(s2));
    if (!strcmp(s1,s2))
    printf("The strings are equal \n");
    strcat(s1,s2);
    printf("%s \n",s1);
}
```

Output:

```
enter the first string
Hello
enter the second string
Hello
The length of the first string is 5 and second is 5
The strings are equal
Hello Hello
```

UNIT – 6
POINTERS

6.1 Introduction:-

A pointer is a variable that represents the location (rather than the value) of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have number of useful applications.

For example, Pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other function to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.

Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements. Moreover, pointers provide a convenient way to represent multidimensional array to be replaced by a lower dimensional array pointers. This feature permits a group of strings to be represented within a single array, through the individual strings may differ in length.

6.2: Pointer Declaration:-

A pointer is a variable which holds the memory address of an another variable. Sometimes, only with the pointer the complex data type can be declared and accessed in an as easy way.

The pointer has the following advantages:-

- supports dynamic allocation routines
- improves the efficiency of certain routines

A pointer contains a memory address. Most commonly, this address is the location of another variable where it has been stored in memory. If one variable, contains the address of another variable, then the first variable is said to point to the second.

Sometimes, pointer is the only technique to represent and to access the complex data structures in an easy way. In C, the pointers are distinct such as integer pointer, floating point number pointer, character pointer etc. The pointer variable consists of two parts as the pointer operator and the address operator.

Pointer operator:-

A pointer operator can be represented by the combination of * (asterisk) with a variable. For example, if the variable is an integer type and also declared with variable that means the variable is of type "pointer to integer". In other words it will be used in the program indirectly to access the value of one or more integer variables.

For example,

```
int *ptr;
```

where ptr is a pointer variable which holds the address of an integer data type.

All pointer variables must be declared before it may be used in our C programs like other variables. When a pointer variable is declared, the variable name must be preceded by an *. This identifies that variable is a pointer.

The general format of the pointer declaration

data – type * Pointer variable;

where data-type is a type of pointer variable such as integer, character, floating point number variable etc. The pointer variables are any valid C identifiers and the * must be preceded by the pointer variable.

Some valid pointer declarations:-

```
float   * fpointer  ;
double  * dpoint    ;
char    * mpoint 1  ;
```

The base type of the pointer defines which type of variables the pointer is pointing to. Technically, any type of pointer can point anywhere in memory. All pointer arithmetic is done relative to its base type. So it is important to declare the pointers correctly.

For example,

a character data items may require to store only a byte (8 bits), an integer may require to store a two bytes (16 bits), a floating point member may require to store four bytes (32 bits), a double precision number may required to store particular data items will vary from machine to machine.

Address operator:-

An address operator can be represented by the combination of & (ampersand) with a pointer variable. For example, if the pointer variable is an

integer type and also declared & with pointer variable that means the variable is of type "address of". In other words, it will be used in the program to indirectly access the value of one or more integer variables. The & is a unary operator that returns the memory address of its operand. An unary operator only requires one operand.

For example,

```
m = &ptr;
```

Note that the pointer operator & as on operator that returns the address of the variable following it. Therefore, the preceding assignment statement could be verbalised as "m receives the address of ptr". The operator *, is the complement of &. It is a unary operator that returns the value of the variable located at the address that follows. The operation of *as translating of the phrase "at address". The symbol * represents both the multiplication sign and the "at address". The symbol & represents both bitwise AND and the "address of" sign. When they are used as pointer operators, these operators have no relationship of the arithmetic operators that happen to look the same. Both the pointer operators, & and * here a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

The pointer operator and address operator & and * are member of the same precedence group as the other unary operators such –,++, – –, ,!, size of and cast operator.

Note:-

The group of unary operators have the higher precedence than the group of arithmetic operators and the associative of the unary operators are left to right.

Pointer Expression:

A pointer is a variable data type, so the general rule to assign its value to pointer is same as any other variable data type.

For example,

```
int x,y;
int *ptr1, *ptr2;
```

1)

```
ptr 1 = &x;
```

The memory address of variable x is assigned to a pointer variable ptr1.

(2)

y=*ptr1

The contents of the pointer variable ptr1 is assigned to the variable y, not the memory address.

(3)

ptr1 = &x;

ptr2 = ptr1; /* address of ptr1 is assigned to ptr2*\ the address of the ptr1 is assigned to the pointer variable ptr2. The contents of both ptr1 and ptr2 will be some because these two pointer variables hold the same address.

Some invalid Pointer declaration:-

(1)
```
int x;
int x_pointer;
x_pointer = &x;
```
Error: Pointer declaration must have the prefix of * (unary operator)

(2)
```
float y;
float *y_pointer;
y_pointer = y;
```
Error: while assigned variable to the pointer variable the address operator (&) must be used along with the variable y.

(3)
```
int x ;
char *C_pointer;
C_pointer = &x;
```
Error:- mixed data type is not permitted.

Program:-

A program to display the contents of the pointer.

```
main()
{
  int x;
  int *ptr;
  x=10;
  ptr =&x;
  printf("x=%d and ptr =%x\n", x, ptr);
  printf ("x=%d and *ptr =%\n,x, *ptr);
}
```

In the above program x is an integer variable and the ptr is declared as the pointer to an integer. Initially, value 10 is assigned to the integer variable x. The address of the variable x is assigned to the ptr.

```
ptr  = &x       /* address of is assigned to the variable pointer*\
*ptr = &x       /* the value of x is assigned to the ptr*\
```

Program:-

A program to assign a character variable to the pointer and to display the contents of the pointer.

```
Main()
{
   char x,y;
   char *pointer;
   x ='c'; /*assign character*/
   pointer = &x;
   y = * pointer;
   printf("value of x = %c\n",x);
   printf("pointer value = %c\n",y);
}
```

Output:-

Value of x = c
Pointer value = c

Program:-

A program to display the address and the contents of a pointer variable.

```
#include stdio.h>
main()
{
   int x;
   int *ptr;
   x = 10;
   ptr = &x;
   printf("value of x = %d\n",x);
   printf("contents of ptr = %d\n",*ptr);
   printf("Address of ptr = %d\n",ptr);
}
```

output:

Value of x = 10
Contents of ptr = 10
Address of ptr = 65488

Program:

```
#include<stdio.h>
main()
{
   int x;
   int  *ptr1, *ptr2;
   x=10;
   ptr1 = &x;
   ptr2 = ptr1;
   printf("Value of x = %d\n",x);
   printf("Contents of ptr1 = %d\n",*ptr1);
   printf("Contents of ptr2 = %d\n",*ptr2);
}
```

Output:

Value of x =10
Contents of ptr1 = 10
Contents of ptr2 = 10

Pointer Arithmetic:-

As we discussed above that the pointer is a variable and it holds the memory address of that variable, some arithmetic operations can be performed with pointers. The c language supports four arithmetic operators that may be used with pointers such as:

Addition +
Subtraction –
Incrementation ++
Decrementation – –

Pointers are variables. They are not integers, but they can usually be displayed as unsigned integers. The conversion specifier for a pointer is added and subtracted.

For example,

Ptr++ causes the pointer to be incremented but not by1.
ptr– – causes the pointer to be decremented not by 1.

The following program segment illustrates the pointer arithmetic.

The integer value would occupy bytes 2000 and 2001.

```
Int value ,*ptr;
Value = 120 ;
Ptr = & value;
Ptr++;
Printf("%u\n",ptr);
```

The above C program segment displays 2002.

The pointer ptr is originally assigned the value 2000. The incrementation, ptr++, increments the number of bytes of the storage class for the particular machine. If the system used four bytes to store an integer, then ptr++ would have resulted in ptr being equal to 2004.

The general rule about pointer arithmetic is that pointer performs the operation in bytes of the appropriate storage class.

Program:-

A program to display the memory address of a variable using pointer before incrementation and after incrementation.

```
/*Pointer arithmetic*\
# include<stdio.h>
main()
{
    int value ;
    int *ptr;
    value = 120;
    ptr = & value;
    printf("Memory address before incrementation = %u\n", ptr);
    ptr++;
    printf("Memory address after incrementation = %u\n", ptr);
}
```

Output:-

```
Memory address before incrementation = 65486
Memory address after incrementation = 65488
```

Program:-

A program to display the memory address of a variable using pointer before decrementation and after decrementation:-

```
/*Pointer arithmetic*\
#include<stdio.h>
main()
{
    float value ;
    float *ptr;
    value = 120.00;
    ptr = & value;
    printf("Memory address = %u\n", ptr);
    ptr-- ;
    printf("Memory address after decrementation = %u\n", ptr);
}
```

Output:-

```
Memory address = 65488
Memory address after decrementation = 65484
```

Whenever a pointer is incremented, its points to the memory location of the next element of its base type and its points to the location of previous element on each decrementation.

In the case of pointers to characters, this often produces what appears to be the "normal" arithmetic. However, all other pointers increase or decrease the length of the data type they point to.

For example, assume 1 byte characters and 2 byte integers when a character pointer is incremented, its value increases by 1; however when an integer pointer is incremented, its value increases by 2. The reason for this is that each time a pointer is incremented or decremented relative to its length of its base type so that it always points to the next element.

All pointer arithmetic's are done relative to the base type of the pointer so that the pointer is always pointing to the appropriate element of the base type the pointers are not limited to only incrementing or decrementing. The pointer variable may be added or subtracted to or from integers.

For example,

```
Ptr = ptr+9;
```

Makes ptr to point to the ninth element of ptr type beyond the element it currently points to.

Program:-

A program to display the memory address of a variable using pointer and to add with an integer quantity with pointer and to display the contents of the pointer.

```
/*Pointer arithmetic*\
# include<stdio.h>
main()
{
    int x ;
    int *ptr1, ptr2;
    x = 10;
    ptr1 = &x;
    ptr2 = ptr1+6;
    printf("Value of x = %d\n",x);
    printf("Contents of ptr1=%d\n",*ptr1);
    printf("Address of ptr1 = %u\n",ptr1);
    printf("Address of ptr2 = (ptr1+6)=%u\n",ptr2);
    printf("Contents of ptr2 = %d\n",ptr2);
}
```

Output:

```
Value of x =10
Contents of ptr1=10
Address of ptr1 = 65490
Address of ptr2 = (ptr+6) = 65502
Contents of ptr2 = –30
```

The memory address of the pointer variable ptr1 is 65490 and an integer value 6 is added with a pointer ptr2 For one integer the pointer variable takes 2 bytes to store then, the resultant address value is 65502.

No other arithmetic operations are allowed other than the addition and subtraction with pointers on integers. To be specific, pointers are not permitted to perform the following arithmetic operations such as

- to multiply or divide
- to operate the bitwise swift and mask operations
- to add or subtract type float or type double to pointers.

Program:-

A program to display the address of and contents of a pointer variable and to subtract with an integer quantity and to display the address and the contents of the pointer variable.

```
/*Pointer arithmetic*/
# include<stdio.h>
main()
{
    int x ;
    int *ptr1, ptr2;
    x = 10;
    ptr1 = &x;
    ptr2 = ptr–2;
    printf("Value of x = %d\n",x);
    printf("Contents of ptr1=%d\n",*ptr1);
    printf("Address of ptr1 = %u\n",ptr1);
    printf("Address of ptr2 = (ptr1–2)=%u\n",ptr2);
    printf("Contents of ptr2 = %d\n",ptr2);
}
```

Output:-

```
Value of x = 10
Contents of ptr1 = 10
Address of ptr1 = 65490
Address of ptr2 = ptr–2 =65486
Contents of ptr2 = 1319
```

Program:-

A program to display the contents of the pointer variables using arithmetic operation.

```
/*Pointer arithmetic*/
# include<stdio.h>
main()
{
    int x ,y;
    int *ptr
    x = 10;
    ptr = &x;
    printf("Value of x = %d and pointer = %d\n",x,*ptr);
    y=++*ptr;
    printf("Contents of ptr2 = %d\n",ptr2);
}
```

Output:-

Value of x=10 and pointer = 10
Value of y= 11 and pointer = 11

Program:

A program to display the contents and the address of a pointer variable using the different type pf incrementation.

```
/*Pointer arithmetic*/
#include<stdio.h>
main()
{
   int x;
   int  *ptr, *ptr2;
   x=10;
   ptr1 = &x;
   printf("Contents of pointer = %d\n",*ptr);
   *ptr = *ptr+1;                  /*(*ptr++)*/
   y = *ptr;
   printf("Value of y  = %d and pointer = *ptr+1=%d\n", y, *ptr);
   *ptr = *ptr+1;                  /*(*ptr++)*/
   y = *ptr;
   printf("value of y = %d and pointer *ptr+1= %d\n",y,ptr);
   (*ptr) ++;                      /*parentheses are necessary*/
   y = *ptr;
   printf("value of y = %d and pointer = (*ptr)++ = %d\n", y, *ptr);
   ++*ptr
   y = *ptr;
   printf("value of y = %d and pointer = (++*ptr) = %d\n,"y,*ptr);
   ++*ptr
   ptr2 = ptr;
   printf("pointer 1 = %d and pointer 2 = %d \n",*ptr,ptr2);
}
```

Output:

Content of pointer = 10 .
Value of y = 11 and pointer *ptr = *ptr+1 = 11
Value of y = 12 and pointer *ptr+1 = 12
Value of y = 13 and pointer (*ptr)++ = 13
Value of y = 14 and pointer (++*ptr) = 14
Pointer 1=15 and pointer 2=15

The address operator cannot act upon arithmetic expression.

For example

2*(x+y) where x and y declared as a pointers variable

Program:-

A program to display the contents of a pointer variable using an ordinary arithmetic expression and the other is a pointer expression.

```
x = 5*(temp+5)
    Where temp = 3
x = 5 *(3+5)
  =5*(8) = 40.  using the pointer arithmetic
x pointer = & temp;
where temp = 3
   y = 5 *(3+5)
     = 5 * 8 = 40
```

6.3: Pointers and Functions:-

The pointers are very much used in a function declaration. Sometimes only with a pointer a complex function can easily be represented and accessed. The usage of the pointers in a function definition may be classified into two groups such as

1. Call by value.
2. Call by reference

Call by value:-

Whenever a portion of the program is calling a function with a formal argument, the control will be transferred from the main to the calling function and the value of the actual argument is copied into the function. Within the function, the actual value copied from the calling portion of the program may be altered or changed. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back. This type of passing formal arguments to a function is called technically, call by value,

For example, the following program segments illustrate the usage of the call by value.

```
main()
{
    int x,y,z ;
    x = 10;
    y = 20;
```

```
    ________________
    ________________
    funct(x,y);         /*call by value *\
    ________________
    ________________
    z = x+y;   /*z=10+20=30*\
 }
funct (a,b)
int a,b;
{
   int sum;
   a=a*a;      /*New values will not be copied to the main *\
   b=b*b;
}
```

Program:-

A program to display the contents of the variable before and after the function using a call by value.

```
/* call by value *\
# include<stdio.h>
main()
{
    int i ;
    i = 20;
    printf("Value of i before function call = %d\n",i);
    f(i);
    printf("\n");
    printf("Value of i after function call = %d\n",i);
 }
f(i)
int i;
{
    int x;
    x = 5*i;
    printf("Inside the function, value of i= %d)"n,x);
    return(x);
}
```

Output:-

```
Value of i before function call = 20
Inside the function value of i = 100
Value of i after function call = 20
```

Program:-

A program to exchange the contents of the two variables using a call by value.

```
#include<stdio.h>
main()
{
int x,y;
x=100;
y=20;
printf("values before swap() %d \t %d \t %d\n; x, y);
swap(x,y);              /* call by value *\
printf("values after swap () %d \t %d\n", x,y);
}
swap(x,y)     /* values will not be swaped *\
int x,y;
{
    int temp;
    temp = x;
    x =y;
    y =temp;
}
```

Output:-

Value before swap () 100 20
Value after swap () 100 20

Since the above function being declared as call by value, the desired result is not yielding. As a single variable is transferred to the function it protects the value of this variable from alteration within the function. On the other hand, it prevents the altered value being transferred back from the function to the calling portion of the programs.

Call by reference:-

When a function is called by a portion of the program, the actual arguments and the values altered within the function will be returned to the calling portion of a program in the altered form. This is called technically as call by reference, call by address, call by location.

With an argument passed by value, the data item is copied to the function. Thus any alteration made to the data item with in the function is not carried over into the calling routine. When our argument is passed by reference (ie.when a

pointer is passed to the function) the address of data item is passed to the function. The contents of that address can be accessed freely, either within a function or within a calling routine.

Moreover, any change that is made to the data item (ie, the contents of the address) will be recognized in both the function and the calling portion of the program. Thus use pf a pointer as a function arguments permit the corresponding data item to be altered globally from within the function.

For example,

The program segments illustrate the usage of the call by reference.

```
Main()
{
   int x,y,z;
   x=10;
   y=20;
   __________
   __________
  funct(&x,&y);  /*call by reference *\
  z = x+y;       /*z=20+40=60 *\
  __________
  __________
}
funct (a,b)
int *a, *b;
{
   *a = *a + *a;       /*new values will be copied to the main*\
   *b = *b++b;
}
```

Program:-

A program to display the contents of the variable before and after the function using a call by reference.

```
#include<Stdio.h>
{
  int i;
  i = 20;
  Printf("Value of i before function call = %d \n" i);
  f(&i);
```

```
  printf("value of i after function call = %d \n",i);
}
  f(int *i)
{
  int temp, n;
  temp = *i;
  n = 7;
  *i= n + temp;
  printf("Inside the function, value of i = %d\n", * i);
}
```

Output:

Value of i before function call = 20
Inside the function, value of i = 27
Value of i after function call = 27

Program:-

A program to exchange the contents of the two variables using call by reference.

```
#include<stdio.h>
main()
{
  int x,y;
  x = 100;
  y = 20;
  printf("Values before swap() %d \t%d\n", x,y);
  swap(&,&y); /*call by reference *\
  printf("values after swap() %d \t %d\n", x,y);
}

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

Output:-

Values before swap() 100 20
Values After swap() 20 100

Program:-

Program that illustrates the difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference.

```
#include<stdio.h>
funct1 (inta, intv);          /*function prototype *\
funct2 (int*pu, int*pr);      /*function prototype *\
main ()
{
  int u,v;
  u=1;
  v=3;
  printf("Before callint funct1: u =%d\t v = %d\n", u,v);
  funct1(u,v);
  printf("After calling funct1: u = %d\t v=%d\n",u,v);
  printf("Before calling funct2: u = %d\t v = %d \n",u,v);
  funct2(&u,&v);
  printf("After calling funct2:u=%d\tv=%d\n",u,v);
}
funct 1(intu,intv)
{
  u = 0;
  v = 0;
  printf("within funct1: u = %d\t v = %d \n",u,v);
  return;
}
funct 2(int*pu,int*pv)
{
   *pu = 0;
   *pv = 0;
   printf("within funct2; *pu = %d \t *pv = %d\n", *pu,*pv);
   return;
}
```

This program contains two functions namely funct1 and funct2. The first function receives two integer variables as arguments. These variables are originally assigned the values 1 and 3 respectively. The values are changed to 0,0 within funct1. The new values are not recognized in main however, because the arguments were passed by value, and any changes to the arguments are local to the function in which the changes occur.

Now consider the second function, funct2. This function receives two pointers to integer variables as its arguments. The arguments are identified as pointers by the indirection operators (ie,*) that appear in the argument declaration. In addition, the argument declaration indicates that the pointers contain the addresses of integer quantities.

Within funct2, the contents of the pointer addresses are reassigned the values 0.0. Since the addresses are recognized in both funct2 and main the reassigned values will be recognized within main after the call to funct2. Therefore, the integer variables u and v will have their values changed from 1,3, to 0,0.

Output:

Before calling funct1 : u=1 v =3
Within funct1 u=0 v=0
After calling funct1: u = 1 v = 3
Before calling funct2: u =1 v =3
Within funct2 : u = 0 v = 0
After calling funct2: u=0 r=0

Note that the values of u and v are unchanged within main after the call to funct1, though the values of these variables are changed within main after the call to funct2. Thus the output illustrates the local nature of the alterations within funct1 and the global nature of the alterations within funct2.

Also note that the function prototype
funct2 (int*pu1 int*pv);

The items inside parentheses identify the arguments as pointers to integer quantities. The pointer variables pu and pv have not been declared elsewhere in main. This is permitted in the function prototype, however because pu and pv are dummy arguments rather than actual arguments. The function declaration could also have been written without any variable names as

funct2(int *, int*);

Here u and v are accessed indirectly, by referencing the contents of the addresses represented by the pointer pu and pv. This is necessary because the variables u and v are not recognised as such within funct2.

Program:-

Analyzing a line of text:-

Suppose we wish to analyze a text by examining each of the characters and determining into which of several different categories it falls. In particular, suppose we count the number of vowels, consonants, digits, white space characters and "other" characters(Punctuation, operators, brackets, etc.) This can easily be accomplished by reading in a line of text, storing it in a one-dimensional character array, and then analyzing the individual array element. An appropriate counter will be incremented for each character. The value of each counter (number of vowels, number of consonants, etc) can then be written out often all of the characters have been analyzed.

Let us write a complete C program that will carry out such an analysis. To do so, we first define the following symbols.

Line	– an 80 element character array containing the line of text
Vowels	– an integer counter indicating the no.of vowels.
Constants	– an integer counter indicating the no.of consonants
Digits	– an integer counter indicating the no.of digits
Whitespace	– an integer counter indicating the no.of white space characters (blank spaces or tabs.)
Other	– an integer counter indicating the no/ of characters that do not fall into any of the preceding categories

```
\*count the no.of vowels, consonants, digits, white space characters and "other"
characters in a line of text */

    #include<stdio.h>
    #include<ctype.h>
    scan_line(char line[ ], int*pv, int*pc, int*pd, int*pw, int*po);
    main()
    {
        char line[80];              /*line of text *\
        int vowels = 0;             /*vowel counter*\
        int consonants =0;          /*consonant counter *\
        int digits = 0;             /*digit counter *\
        int whitespace = 0;         /*whitespace counter *\
        int other = 0;              /*remaining character counter *\
        printf("Enter a line of text below:\n");
        scanf("%[^\n]", line);
        scan_line(line, &vowels, &consonants, &digits, &whitespace,& other):
```

```
    printf("\n No.of vowels:%d", vowels);
    printf("\n No.of constants: %d", constants);
    printf("\n No.of digits: %d", digits);
    printf("\n No.of whitespace characters: %d", whitespace);
    printf("\n No.of other characters: %d", other);
}
    scan_line(char line[ ], int *pv, int*pc, int *pd, int*, int*pw, int*po);
        /*analyze the characters in a line of text*\
    {
        char c;                 /*uppercase character*\
        int count =0;           /* character counter*\
        while ((c = to upper (line[count])) ! = '\0')
        {
                if(c='A' || c=='E' || c == 'I' || c=='0' || c=='u');
                ++*pv; /*vowel *\
                else if(c>='0' && c<='9')
                        ++*pd;              /*digit *\
                        else if(c==' '||c ==' \t')
                            ++*pw;          /*whitespace *\
                            else
                              ++*po;

                ++count;
            }
                return;
        }
```

In the function prototype for scan_line the array argument, line, is not preceded by an ampersand, since arrays are, by definition, pointers. Each of the remaining arguments must be preceded by an ampersand so that its address, rather than its value, is passed to the function.

Now consider the function scan_line. All of the formal arguments, including line, are pointers. However, line is declared as an array whose size is unspecified, whereas the remaining arguments are specially declared as pointers. It is possible to declare line as a pointer rather than an array. Thus, the first line of scan_line could have been written as

```
scan_line (char*line, int*pv, int*pc, int*pd, int*pw, int*po)
```

rather than as shown in the program listing.

Enter a line of text below:

Personal computers with memories in excess of 4096 kB are now quite common.

No.of rowels: 23
No.of consonants: 35
No.of digits: 4
No.of whitespace characters: 12
No.of other characters : 1

6.4 Pointers and one Dimensional arrays:-

An array name is really a pointer to the first element in the array. Therefore, if x is one dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. Moreover, the address of the second array element can be written as either &x[i] or as(x+1) and so on. In general, the address of array element (i+1) can be expressed as either &x[i] or as (x+i).

Thus we have two different ways to write the address of any array element:-

1. We can write the actual array element. Preceded by an ampersand.
2. We can write an expression in which the subscript is added to the array name.

In the second case, the expression (x+i), x represents an address, whereas i represents an integer quantity.

Moreover, x is the name of an array whose elements may be characters, integers, floati ৢ. Point quantities, etc. though all of the array elements must be of the same data type. Thus, we are not simply adding numerical values. Rather, we care specifying an address of the first array element. Or, in simpler terms, we are specifying a location that is array elements beyond the first. Hence, the expression (x+i) is a symbolic representation for an address specification rather than an arithmetic expression.

Note:- The no.of memory cells associated with an array element will depend upon the data type of the array as well as the particular computer's architecture.

When writing the address of an array element in the form (x=i), however, you need not be concerned with the no.of memory cells associated with each type of array element; the C compiler adjusts for this automatically. You must specify only the address of the first array element (ie. The name of the array) and the

number of array elements beyond the first (ie. a value for the subscript). The value of i is sometimes referred to as an **offset** when used in this manner.

Since & x[i] and (x+i) both represent the address of the ith element of x, it would seen reasonable that x[i] and *(x+i) both represent the contents of that address, (ie). the value of the ith element of x.

The program segment that illustrates the relationship between array elements and their addresses.

```
#include<stdio.h>
main()
{
   static int x[10] = {10,11,12,13,14,15,16,17,18,19};
   int i;
      for(i=0; i<9; ++i)
      {
         printf("\n:= %d x[i]=%d *(x+i)=%d",i,x[ i ],*(x+i));
         /*display an array element *\
         printf("& x[i]=%x  x+i =%x",&x[ i ],(x+i));
       }
}
```

This program defines a one-dimensional, 10 element integer array x, whose elements are assigned the values 10,11,....9. The value of each array element is specified in two different ways, as x[i] and as *(x+i).

Similarly, the address of each array element is specified in two different ways as &x[i] and as *(x+i).

Output:-

```
i = 0   x[ i ] =10     *(x+i) = 10    & x[ i ] = 72   x+i = 72
i = 1   x[ i ] =11     *(x+i) = 11    & x[ i ] = 74   x+i = 74
i = 2   x[ i ] =12     *(x+i) = 12    & x[ i ] = 76   x+i = 76
i = 3   x[ i ] =13     *(x+i) = 13    & x[ i ] = 78   x+i = 78
i = 4   x[ i ] =14     *(x+i) = 14    & x[ i ] = 7A   x+i = 7A
i = 5   x[ i ] =15     *(x+i) = 15    & x[ i ] = 7C   x+i = 7C
i = 6   x[ i ] =16     *(x+i) = 16    & x[ i ] = 7E   x+i = 7E
i = 7   x[ i ] =17     *(x+i) = 17    & x[ i ] = 80   x+i = 80
i = 8   x[ i ] =18     *(x+i) = 16    & x[ i ] = 82   x+i = 82
i = 9   x[ i ] =19     *(x+i) = 19    & x[ i ] = 84   x+i = 84
```

Note:- The addresses are assigned automatically by the compiler.

When assigning a value to an array element such as x[i], the left side of the assignment statement may be written as either x

When assigning a value to an array element such as x[i], the left side of the assignment statement may be written as either x[i] or as *(x+i). Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element.

On the other hand, it is sometimes necessary to assign an address to an identifier. In such situations, a pointer variable must appear on the left side of the assignment statement. It is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as x, (x+i) and & x[i] cannot appear on the left side of an assignment statement. Moreover, the address of an array cannot arbitrary be altered so that expressions such as ++x are not permitted.

Eg:- Consider the segment of the C program shown below:-

```
#include<Stdio.h>
main( )
{
   int line[80];
   int *pl
  ___________
  ___________
  /* assign values *\
  line[2] = line[1];
  line[2] = *(line+1);
   (line+2) = (line[1]);
  *(line+2) = *(line+1);
  /* assign addresses*\
  pl=&line[ i ]
  pl=&line+I;
}
```

Each of the first assignment statements assigns the value of the second array element (ie. line [1]) to the third array element (line[2]). Thus, the four statements are all equivalent.

The last two assignment statements each assigns the address of the second array element to the pointer pl. Note that the address of one array

element cannot be assigned to some other array element. Thus we cannot write a statement such as

```
&line[ 2 ] = &line [ i ]
```

on the other hand, we can assign the value of one array element to another through a pointer if we wish.

eg:-

```
pl = &line[ i ];
line[2] = *pl;
```

or

```
pl = line+1;
*(line+2) = *pl;
```

If a numerical array is defined as a pointer variable, the array elements cannot be assigned initial values.

Therefore, a conventional array definition is required if initial values will be assigned to the elements of a numerical array. However, a character-type pointer variable can be assigned an entire string as a part of the variable declaration. Thus, a string can conveniently be represented by either a one-dimensional character array or a character pointer.

Example:-

Shown below is a simple C program in which two strings are represented as one-dimensional character arrays.

```
#include<stdio.h>
char x [ ]; “This string is declared externally \n\n”;
main( )
{
    static char y[ ] : “This string is declared within main”;
    printf(“%s:, x);
    printf(“%s”, y);
}
```

The first string is assigned to the external array x[ ]. The second string is assigned to the static array y[ ]. Which is defined within main. This second definition occurs within a function; therefore y[ ] must be defined as a static array so that it can be initialized.

Here is a different version of the same programs. The strings are now assigned to pointer variables rather than to one dimensional arrays.

```
#include<stdio.h>
char *x="This string is declared externally \n\n";
main()
{
  char*y = "This string is declared within main";
  printf("%s",x);
  printf("%s",y);
}
```

The external pointer variable x points to the beginning of the first string. Whereas the pointer variable y, declared within main, points to the beginning of the second string. Note that y can bow be intialised without being declared static.

Execution of either program produces the following output.

```
This string is declared externally
This string is declared within main
```

Dynamic Memory Allocation: -

Since an array name is actually a pointer to the first element within the array, it should be possible to define the array as a pointer it should be possible to define the array as a pointer variable rather than as a conventional array. Syntactically the two definitions are equivalent. However, a conventional array definition results in a fixed block of memory being reserved at the beginning of program execution. Whereas this does not occur if the array is represented in terms of a pointer variable. Therefore the use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as dynamic memory allocation. Generally the Malloc library function is used for this purpose.

Example:-

Suppose x is a one-dimensional, 10 element array of integers. It is possible to define x as a pointer variable rather than an array. Thus, we can write

```
    int *x;
rather than
   int x[10];
or
#define SIZE 10
```

```
int x[size];
```

However, x is not automatically assigned a memory block when it is defined as pointer variable, though a block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array.

To assign sufficient memory for x we can make use of the library function malloc, as follows.

```
x = (int *) malloc (10*size of (int));
```

This function reserves a block of memory whose size (in bytes) is equivalent to 10 integer quantities. As written, the function returns a pointer an integer. This pointer indicates the beginning of the memory block. In general, the type cast preceding malloc must be consistent with the data type of the pointer variable. Thus , if y were defined as a pointer to a double – precision quantity and we wanted enough memory to store 10 double – precision quantities, we would write

```
Y = (double*) malloc (10 *size of (double));
```

If the declaration is to include the assignment of initial values, however, then x must be defined as an array rather than a pointer variable.

For example,

```
int x [10] = {1,2,3,4,5,6,7,8,9,10};
or
int x [ ] = {1,2,3,4,5,6,7,8,9,10};
```

Program:-

Reordering a List of Numbers:-

To illustrate the use of pointers with dynamic memory allocation

/* reorder a one-dimensional, integer array from smallest to largest, using pointer notation*/

```
#include<stdio.h>
#include<std1:b.h>
recorder(intn, int*x)
main()
{
    int i,n,*x;
    /* read in a value for n *\
```

```
        printf("\n How many numbers will be entered?");
        scanf("%d"&n);
        prinstf("\n");
        /*allocate memory*\
        x = (int x) malloc (n* size of (int));
        /*read in the list of numbers*\
        for (i=0;i<n; ++i)
        {
            printf("I = %d x =", i +1);
            scanf("%d",x+i);
        }
        /*reorder all array elements*\
        reorder (n,x);
        /*display the reordered list of numbers*\
        printf("\n\n Reordered list of numbers:\n\n";
        for(i=0;i<n;++i)
        printf("i= %d x=%d\n", i+1,*(x+i)
}
recorder(intn,intx) /*rearrange the list of numbers*\
{
    int i, item, temp;
    for(item=o ;item<n – 1; ++item)
    /*find the smallest of all remaining items*\
    for(i=item+1; i<n; ++i)
   if(*(x+i)<*(x+item))
   {
   /*interchange two items*\
   temp =*(x+item);
   *(x+item) = *(x+i);
   *(x+i)=temp;
    }
return;
}
```

In this program, the integer array is defined as a pointer to an integer. Memory is initially assigned to the pointer variable via the malloc library function. Elsewhere in the program, pointer notation is used to process the individual array elements. For example, the function prototype now specifies that the second argument is a pointer to an integer quantity rather than an integer array. This pointer will identify the beginning of the integer array.

The scanf function now specifies the address of the ith element as x+i rather than &x[i]. Similarly, the printf function now represents the value of the ith element as *(x+i) rather than x[i]

Within the function recorder, we see that the second formal argument is now defined as a pointer variable rather than an integer array. This is consistent with the function prototype. In particular notice each reference to an array element is now written as the contents of an address. Thus x[i] is not written as * (x+i) and x[item] is now written as *(x+item). The compound if statement can be viewed as a conditional interchange involving the contents of two different addresses, rather than an interchange of two different elements within a conventional array.

An important advantage of dynamic memory allocation is the ability to reserve as such memory as may be required during program execution, and then release this memory when it is no longer needed. Moreover, this process may be repeated many times during execution of a program.

Operations on Pointer:-

An integer value can be added to an array name in order to access an individual array element. The integer value is interpreted as an array subscript it represents the location of the desired array element relative to the first element in the array. This works because all of the array elements are of the same data type, and therefore each array element occupies the same number of memory cells (ie) the same no.of bytes or words. The actual number of memory cells that separate the two array elements will depend on the data type of the array, though this is taken care of automatically by the compiler.

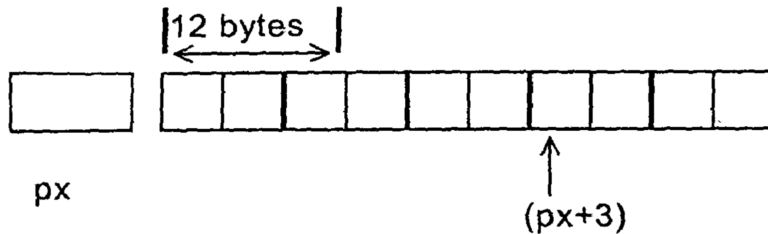
This concept can be extended to pointer variables in general. Thus an integer value can be added to or subtracted from a pointer variable, though the resulting expression must be interpreted very carefully.

For example,

px is a pointer variable that represents the address of some variable x. we can write expressions such as ++px, – –px, (px+3), (px+i) and (px–i) where i is an integer variable. Each expression will represent an address that is located some distance from the original address represented by px. The exact distance will be the product of the integer quantity and the no.of bytes or words associated with the data item to which px points.

For example,

px points to an integer quantity, and each integer quantity requires two bytes of memory. Then the express (px+3) will result in an address that is 6 bytes beyond the integer to which px points 6 bytes beyond the integer to which px points



It should be noted that this new address will not necessarily represent the address of another data item. Particularly if the data item stored between the two addresses involve different data types.

Program:-

```
#include<Stdio.h>
main()
{
  int *px;      /*pointer to an integer*\
  int i = 1;
  float f = 0.3
  double d = 0.005;
  char c = '*';
  px = &;
  printf("values: i = %i f =%f d = %f c = %c \n\n, i,f,d,c);
  printf("Addresses: &i = %x  &f =%x &d = %x &c = %x \n\n", &i,&f,&d,&c);
  printf("Pointer values : px=%x px+1=%x px+2=%x px+3=%x",
                                        px,px+1,px+2,px+3);
}
```

Output:

Values: i = 1 f = 0.300000 d = 0.005000 c = *

Addresses: & i = 117E & f = 1180 & d = 1186 & c = 118E

Pointer values: P_x = 117E P_{x+1} = 1180 P_{x+2} = 1182 P_{x+3} = 1184

The first line displays the values of the variables, and the second line displays the addresses as assigned by the compiler. Note that the no of bytes associated with each data item is different. Thus the integer value represented by i requires two bytes (specially, addresses 117E and 117F) The floating point value represented by f appears to be assigned 6 bytes (addresses 1180 through 1185) though only 4 bytes (addresses 1180 through 1183) are actually used for

this purpose. (Compilers allocate memory space according to their own rules). However 8 bytes are required for the double precision value represented by d (addresses 1186 through 1180) And finally, the character represented by c begins in address 118E only one byte is required to store this single character, though the output does not indicate the no of bytes between this character and the next data items.

The third line of output, contains the addresses represented by the pointer expressions. Clearly, Px represents the address of; (ie 117E) because px = &; However, px+1 moves over only two bytes, to 1180 and px+2 moves over another two bytes to 1182 and so on. The reason is that px points to an integer quantity and integer quantities each require two bytes with this particular C compiler.

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of words or bytes separating the corresponding array elements.

Program:

```
# include < stdio.h >
main ()
{
   int *px, *py;          /* integer pointers */
   Static int a [b]  =  {1,2,3,4,5,6};
   px = & a [0];
   py = &a [5];
   printf ("px = %x py = %x", px, py);
   printf ("\n\n py – px = %x",py – px);
}
```

Output:

```
px = 52        py = 5c
py – px = 5
```

The first line indicates the address of a [0] which is 52 and a [5] as 5c. The difference between these two hexadecimal numbers is 10 (when converted to decimal). Thus a [5] is stored at an address which is 10 bytes beyond the address of a [0]. Since each integer quantity occupies two bytes, the difference between py and px is 10/2 = 5.

Pointer variables can be compared provided both variables are of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparisons can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (which is usually expressed as **NULL** when used in this manner.

Example:

Suppose px and py are pointer variables that point to elements of the same array. Several logical expressions involving these two variables are shown below. All the expressions are syntactically correct.

```
(px < py)
(px > = py)
(px == py)
(px ! = py)
(px == NULL)
```

These expressions can be used as any other logical expression. For example,

```
if (px < py)
  printf ("px < py");
else
   printf ("px > py");
```

Expressions such as (px < py) indicate whether or not the element associated with px is ranked ahead of the element associated with py. (ie whether or not the subscript associated with * py).

The operations discussed previously are only operations that can be carried out on pointers. These permissible operations are summarized below.

1. A pointer variable can be assigned the address of an ordinary variable.
   eg. pr =&r

2. A pointer variable can be assigned the value of another pointer variable.
   eg. pr = px
   provided both pointers point to objects of the same data type.

3. A pointer variable can be assigned a null (zero) value
   eg. rp = NULL, where NULL is a symbolic constant that represents the value 0.

4. An integer quantity can be added to or subtracted from a pointer variable.
   eg. pr + 3, + + pr, ...

5. One pointer variable can be subtracted from another provided both pointers point to elements of the same array.

6. Two pointers can be compared provided both pointers point to elements of the same data type.

Other Arithmetic operations on pointers are not allowed. Thus, a pointer variable cannot be multiplied by a constant two pointer variables cannot be added and so on. Also, an ordinary variable cannot be assigned an arbitrary address (ie an expression such as & x cannot appear on the left side of an assignment statement.

6.5. Pointers and Multidimensional Arrays:

Since a one–dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), it is reasonable to expect that a multidimensional array can also be represented with an equivalent pointer notation.

A two – dimensional array, for example, is actually a collection of one – dimensional arrays. Therefore, we can define a two – dimensional array as a pointer to a group of contiguous one –dimensional arrays/ This a two dimensional array declaration can be written as

data type (* ptvar) [expression 2];
rather than

data type array [expression 1] [expression 2];
This concept can be generalized to higher dimensional arrays.

(ie) data – type (*ptvar) [expression 1] [expression 2] [expression 3] ...[expression n] ; replaces
data type array [expression 1] [expression 2] ..... [expression];

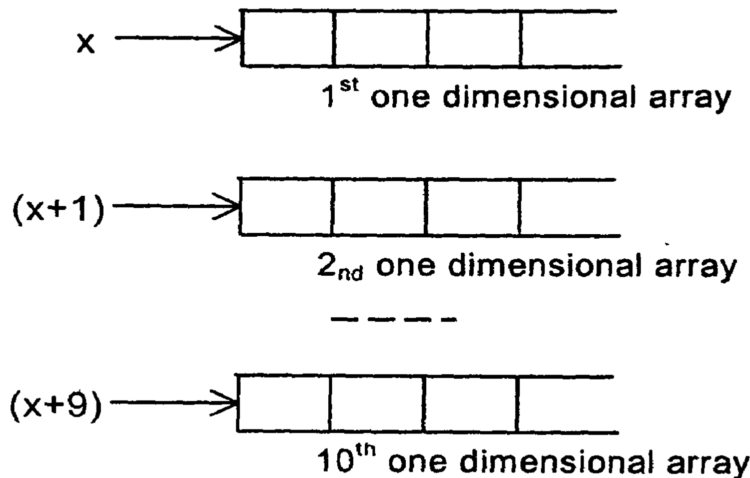
In these declarations **data–type** refers to the data type of the array, **ptvar** is the name o the pointer variable, **array** is the corresponding array name, and **expression 1, expression 2 ....expression n** are positive – valued integer expressions that indicate the maximum no of array elements associated with each subscript.

Example:

Suppose x is a two dimensional integer array having 10 rows and 20 columns. We can declare x as

```
int (*x) [20];
rather than
        int x  [10] [20];
```

In the first declaration, x is defined to be a pointer to a group of contiguous, one dimensional, 20 – element integer arrays. Thus, x points to the first 20 – element array, which is actually the first row (ie row 0) of the original two dimensional array. Similarly, (x+1) points to the second 20 – element array, which is the second row (row 1) of the original two dimensional array, and so on as illustrated. Below



Now consider a three–dimensional floating – point array t.
This array can be defined as

```
float (*t) [20] [30];
```

rather than

```
float t [10] [20] [30];
```

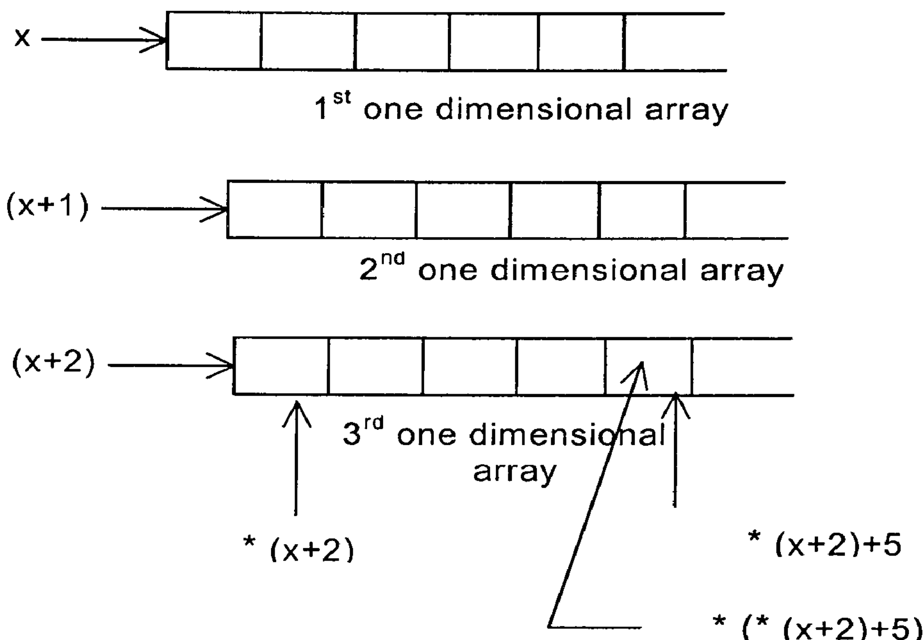
Example:

Suppose x is a two dimensional integer array having 10 rows and 20 columns. The item in row 2, column 5 can be accessed by writing either.

X [2] [5] or * (* (x+2)+5)

First, note that (x+2) is a pointer to row 2. This * (x+2) refers to the entire row. Since row 2 is a one dimensional array * (x+2) is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence (*(x+2)+5) is a pointer to element 5 (ie 6th element) in row 2. Thus * (* (x+2)+5) refers to the item in column 5 of row 2, which is * [2] [5].

This relationships are illustrated below.



Program: Adding two tables of Numbers:

In this program each two dimensional arrays, is defined as an array of pointers to a set of one–dimensional integer arrays.

/* calculate the sum of the elements in two tables of integers */

/* each 2 – dimensional array is processed as an array of pointers to a set of 1–dimensional integers arrays/*

```
# include < stdio.h >
# include < stdio.h >
# define MAXROWS 20
/* function Prototypes */
readinput (int * a [MAXROWS]. intnrows, int ncols);
compute sums (int * a [MAXROWS],int * b[MAXROWS],int *c[MAXROWS]
                                        int nrows,int ncols);
write output (int *c[MAXROWS], int nrows, intn cols);

main ()
{
   int row,nrows,ncols;
   int *a[MAXROWS], * b[MAXROWS], *c[MAXROWS] ;
                                   /* pointer definitions */
   printf ("How many rows?");
   Scanf ("%d", &n rows);
   Printf ("How many columns?");
   Scanf ("%d", & n cols);
   /* allocate initial memory */
   for (row = 0; row < n rows; ++row)

   {
        a [row] = (int *) malloc (ncols * size of (int));
        b [row] = (int *) malloc (ncols * size of (int));
```

```
            c [row] = (int *) malloc (ncols * size of (int));
        }
        printf ("\n\n First table: \n");
        readinput (a,nrows,ncols);
        compute sums (a,b,c,nrows,ncol);
        printf ("\n\n sums of the elements: \n\n");
        write output (c,nrows,ncols);
    }
    readinput (int *a[MAXROWS], intn, intn)
    \* read in a table of integers */
        {
            int row, col;
            for (row = 0; row < n; ++ row)
            {
                printf (") n Enter data for row no. % 2d \n", row +1);
                for (col = 0;  col < n ; ++ col)
                Scanf ("%d", (* (a+row) +col));
            }
            return;
        }
    computesums (int * a [MAXROWS], int * b [MAXROWS],
                     int * c [MAXROWS], int n, intn)
    /* add the elements of two integer tables */
    {
        int row, col;
        for (row = 0, row <n; ++row)
        for (col = 0; col <n; ++col)
            * (*(c+row) + col) = * (* (a +row)+col)+
                                 * (* (b +row) +col);
        return;
    }
    write output (int *a [MAXROWS],intn, intn)
    /* write out a table of integers */
      {
            int row col;
            for (row = 0; row < n; ++row)
      {
            for (col = 0; col < n; ++col)
               printf ("% *d", * (*(a + row) + col));
            printf ("\n");
      }
      return;
    }
```

Output:

```
How many rows?       3
How many columns?  4

First table:
Enter data for row no.1
1       2       3       4
Enter data for row no. 2
```

```
5       6       7       8
Enter data for row no. 3
9       10      11      12
Second table:
Enter data for row no. 1
10      11      12      13
Enter data for row no. 2
14      15      16      17
Enter data for row no. 3
18      19      20      21
Sum of the elements
11      13      15      17

19      21      23      25

27      29      31      33
```

In this program a,b,c are each defined as an array of pointers to integers. Each array has a maximum of MAXROWS elements, since each element of a,b,c is a pointer, we must provide each pointer with enough memory for each row of integer quantities, using the malloc function.

Consider the first memory allocation (ie)

A [row] = (int*) malloc (ncols * size of (int));

In this statement a [0] points to the first row, a [1] points to the second row and so on. Thus each array element points to a block of memory large enough to store one row of integer quantities (ncols integer quantities). Similar memory allocations are written for the other two arrays.

In readinput each array element is referred as
Scanf ("%d", (* (a+row) col)); (1)

Similarly, the addition of two array elements with compute sums is written as

* (*(c + row)+col) = * (*(a+row)+col)+* (*(b+row)+col); (2)

and the first printf statement within writeoutput is
printf ("%*d", * (*(a+row) + col)); (3)

We would of course, have used the more conventional notation within the functions.

Thus, in readinput we could have written
Scanf ("%d", a [row] [col]); instead of (1).

Similarly, in computesums we could have written
C [row] [col] = a [row] [col] + b [row] [col]
Instead of (2)
and in writeoutput we could have written
Printf ("%*d", a [row] [col]); instead of (3)

6.6. Arrays of Pointers:

A multidimensional array can be expressed in terms of an **array of pointers** rather than a **pointer to a group of contiguous arrays.**

In such situations the newly defined array will have one less dimension than the original multidimensional. Each pointer will indicate the beginning of a separate (n–1) dimensional array.

In general terms, a two dimensional array can be defined as a one–dimensional array of pointers by writing.

data–type * array [expression 1];
rather than
data–type array [expression 1] [expression 2];

Similarly, an n – dimensional array can be defined as an (n–1) dimensional array of pointers by writing

data – type * array [expression 1] [expression 2]
[expression n –1];

rather than
data – type array [expression 1] ...[expression n];

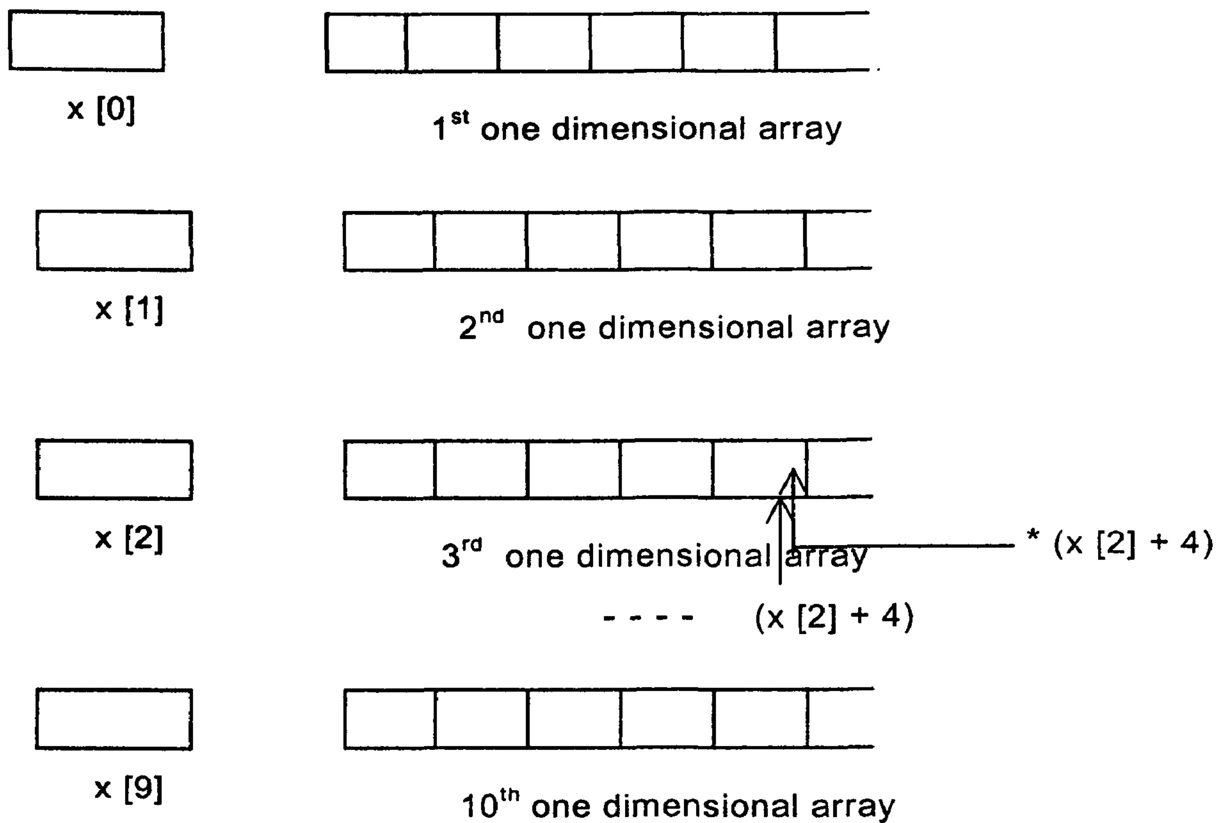
Note that the array name and its preceding asterisk are not enclosed in parentheses in this type of declarations.

Also the last expression is omitted when defining an array of pointers, whereas the first expression is omitted when defining a pointer to a group of arrays.

Example:

Suppose x is a two–dimensional integer array having 10 rows and 20 columns then we can define x as a one–dimensional array of pointers by writing

int * x [10];



An individual array element, such as x [2] [4], can be accessed by writing * (x[2] +4)

Here in this expression x [2] is a pointer to the first element in row 2, so that x[2] + 4 points to element 4. (actually the fifth element) within row 2. The object of this pointer * (x [2] +4) therefore refers to x[2] [4].

Example:

Program to add two tables of Numbers in which each 2 –dimensional array is to be represented as an array of pointers.

This program is left as an exercise.

Program: Reordering a List of strings:

Let us now approach this problem using a one–dimensional array of pointers,, where each pointer indicates the beginning of a string. The string interchanges can now be carried out simply by reassigning the pointers as required.

```
/* sort a list of strings into alphabetical order using an array of pointers */
      # include <stdio.h >
      # include < stdio.h >
      # include < string.h >

      reorder (int n, char * * [ ]);
      main ()
      {
         int i,n = 0;
         char * x [10];
         printf ("Enter each string on a separate line below \n\n");
         printf ("Type \' END\' when finished \n\n);
         /* read in the list of strings */
         do
             {
                /* allocate memory */
                x [n] = (char *) malloc (12 * size of (char));
                printf ("string %d;", n+1);
                Scanf ("%&", s [n]);

             }
             while (stramp (x[n+1], "END"));
             /* reorder the list of strings */
             recorder (– – n, x);
             /* display the reordered list of strings */
             printf ("\n\n Reordered List of strings: \n);
             for (i = 0; i < n; ++i)
                printf ("\n string %d; %S"; +1, x[i]);
     }
     reorder (int, char* * [ ]) /* rearrange the list of strings*/
     {
        char * temp;
        int; , item;
        for (item = 0; item < n–1; ++item)
        /* find the lowest of all remaining strings */
        for (i = item + 1; ; < n; + + i)
        if (stramp (x [item], x [i] 70)
        {
            /* interchange the two strings */
            temp = x [item];
            x [item] = x [1];
            x [I] = temp;
      }
      return;
    }
```

Output:

```
Enter each string on a separate line below
Type `END' when finished
String 1:    Kepler
String 2:    Descartes
String 3:    Fermat
String 4:    Leibnitz
String 5:    Euler
String 6:    Gauss
String 7:    Boole
String 8:    Hilbert
String 9:    Niels
String 10:   Abel
String 11:   END

Reordered List of strings:
String 1:    Abel
String 2:    Boole
String 3:    Descartes
String 4:    Euler
String 5:    Fermat
String 6:    Gauss
String 7:    Hilbert
String 8:    Kepler
String 9:    Leibnitz
String 10:   Niels
```

If the elements of an array are string pointers a set of initial values can be specified as a part of the array declaration. In such cases the initial values will be strings, where each string corresponds to a separate array element. However, that an array must be declared **static** if it is initialized within a function.

An advantages to this scheme is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional array. Thus, if the initial declaration includes many strings and some of them are relatively short, there may be a substantial savings in memory allocations. Moreover, is some of the strings are particularly long, there is no need to worry about the possibility of exceeding some maximum specified string length. Arrays of this type are other referred to as ragged arrays.

Example:

The following array declaration appears within a function.

```
Static char * names [10] = {
                            "Kepler",
                            "Descartes",
                            "Fermat",
                            "Leibnitz",
```

```
"Euler",
"Gauss",
"Boole",
"Hilbert",
"Niels",
"Abel"
};
```

Program: Displaying the Day of the year:

Let us develop a program that will accept 3 integer quantities, indicating the month, day and year, and then display the corresponding day of the week, the month, the day and the year in more legible manner.

For example: suppose we enter 5 24 1997 then the output should be Saturday, May 24, 1997.

The computation can be carried out using the following empirical rules.

1. Enter numerical values for the variables mm,dd,yy, which represent the month,day and year respectively eg.11 4 1977

2. Determine the approximate day of the current year, as n days = (long) (30.42 * (mm – 1) +dd;

3. If mm ==2 (February), increase the value of n days by 1.

4. If mm > 2 and mm < 8 (March, April,May,June,July) decrease the value of n days by 1.

5. Convert the year into the number of years beyond the base data; (ie) yy ==1900. Then test for a leap year as follows: If (yy% 4) == 0 an mm > 2, increase the value of n days by 1.

6. Determine the number of complete 4 year cycles beyond the base data as yy/4. For each complete 4 year cycle, add 1461 to n days.

7. Determine the number of full years beyond the last complete 4 year cycle as yy%4. For each full year, add 365 to n days. Then add 1, because the first year beyond a full 4 – year cycle will be a leap year.

8. It n days > 59 (ie if the data is any day beyond February 28, 1900), decreases the value of n days by 1, because 1900 is not a leap year (Note that the last year of each century is not a leap year, except these years that are evenly divisible by 400. Therefore 1900, the last year of the

nineteenth century, is not a leap year, but 2000, the last year of the twentieth century is a leap year).

9. Determine the numerical day of the week corresponding to the specified data as day = (n days % 7).

Note that day == 1 corresponds either to the base data which is a Monday, or another date that also occurs on a Monday. Hence day = 2 will refer to a Tuesday, day = 3 Saturday and day = 0 will refer to a Sunday.

Note that n days and ncycles are long integer variables whereas all other variables are ordinary integers. Here is an entire C program that will carry out the conversion interactively.

```
/* convert a numerical date (mm dd yyyy) into "day of week, month, day, year"/*
    # include < stdio.h >
    readinput (int * pm, int, *pd, int *py);        /* function prototype */
    int convert (int mm, int dd,  int yy);           /* function prototype*/
    main ()
    {
       int mm, dd, yy;
       int day-of-week; /* day of the week (0 → Sunday,
                                                1 → Monday,
                                                6 → Saturday)*/
       Static char * weekday [ ] : {"Sunday", "Monday", "Tuesday",
                                    "Wednesday", "Thursday",
                                    "Friday", Saturday"};
       Static char * Month [ ]:    {"January", "February", "March",
                                    "April", "May", "June", "July", "August",
                                    "September", October", "November",
                                    "December"};
       /* opening message */
       printf ("Date conversion Routine \n To stop, enter 0,0,0");
       readinput (&mm, &dd, &yy);
       /* convert data to numerical day of week */
       while (mm 70)
       {
           day-of-week = convert (nn,dd,yy);
           printf ("\n %s, %s, %d, %d", Weekday [day-of-week],
                                          month [mm-1], dd, yy);
           readinput (&mm, &dd, &yy);
       }
```

```
}
readinput (int * pm, int * pd, int * py) /* read in the numerical date */
{
   printf ("\n\n Enter mm dd yyyy:");
   Scanf ("%d%d%d", pm,pd,py);
   Return;
}
int convert (int mm, int dd, int yy) /* convert date to numerical day of week
*/
{
   long n days;        /* no of days from start of 1900 */
   long ncycles;       /* no of  4-year cycles beyond 1900 */
   int nyears;         /* no of years beyond last 4-year cycle */
   int day;     /* day of week (0,1,2,3,4,5 or 6) */
   /* numerical conversions */
   yy - = 1900
   n days = (long) (3042 * (mm-1) +dd;
             /* approximate to day of year */
   if (mm == 2) ++n days;   /* adjust for February */
   if ((mm > 2) && (mm < 8)) – n days; /* adjust for March – July */
   if ((yy % 4 = = 0) && (mm > 2)) ++ n days;
                                    /* adjust for leap year */
   ncycles = yy / 4;  /* 4 year cycles beyond 1900 */
   ndays + = ncycles * 1461;       /* add days for 4 year cycles */
   nyears = yy% 4;  /* years beyond last 4 year cycle */
   if (n years > 0  /* add days for years beyond last
       n days + = 365 * n years + 1;      4 year cycle */
   if (n days > 59)   - - n days;     /* adjust for 1900
                                    (NOT a leap year) */
   day = ndays%7;
   return (day);
}
```

Output:

```
Date conversion Routine
To stop, enter 0      0      0
Enter mm dd yyyy   8      15      1945
Wednesday, August 15, 1945
Enter mm dd yyyy   0      0      0
```

6.7. Passing functions to other functions:

A pointer to a function can be passed to another function as an argument. This allows one function to be transferred to another, as though the first function were a variable. Let us refer to the first function as the guest function, and the second function as the host function. Thus, the guest is passed to the host, where it can be accessed. Successive calls to the host function can pass different pointers (ie different guest functions) to the host.

When a host function accepts the name of a guest as an argument, the formal argument declaration must identify that argument as a pointer to the guest function.

The formal argument declaration can be written as

Data-type (* function-name) ()

Where data type refers to the data type of the quantity returned by the guest and function-name is the name of the guest.
The formal argument declaration can be written as

Data-type (* function name) (type 1, type2, ...)

Data type (*function-name) (type 1 arg 1, type 2 arg 2. . . ..)

Where type 1, type 2, ... refer to the data types of the arguments associated with the guest, and arg 1, arg 2,.... refer to the names of the arguments associated with the guest.

The guest function can be accessed within the host by means of the indirection operator. To do so, the indirection operator must precede the guest function name (ie. the formal argument). Both the indirection operator and the guest function name must be enclosed in Parentheses.

(i.e.) * (function-name) (argument 1, argument 2, ... argument n);
where arguments 1, argument 2, ... argument n refer to the arguments that are required in the function call.

The function declaration for the host function may be written as
Funct-data-type funct-name (arg-data-type (*) (type 1, type 2,....),

|⟵ pointer to guest function ⟶|

data types of other funct args);

where funct-data-type refers to the data type of the quantity returned by the host function, funct-name refers to the name of the host-function. arg-data-type refers

to the data type of the quantity returned by the guest function and type 1 and type 2 . . . . refer to the data types of the guest function's argument.

Note that the indirection operator appears in parentheses to indicate a pointer to the guest function. Moreover, the data types of the guest function's argument follow in a separate pair of parentheses, to indicate that they are function arguments.

When full function prototyping is used, the host function declaration is expanded as follows.

Funct-data-type funct-name
(arg-data-type (*pt-var) (type 1 arg 1, type 2 arg 2, ...)

|←—— pointer to guest function ——→|
data types and names of other funct args);

Example:

The segment of a C program is given below. This program consists of two functions: main, process, funct1, funct 2. Note that process is a host function for funct 1 and funct 2. Each of the 3 subordinate functions returns an integer quantity.

```
int process (int (*) (int,int));      /* function declaration (host)*/
int funct 1 (int, int);               /* function declaration (guest)*/
int funct 2 (int, int);               /* function declaration (guest)*/
main ()
{
   int:,i,j;
   ____________
   i = process (funct 1); /* pass funct 1 to process; return a value for i */
   j = process (funct 2); /* pass funct 2 to process; return a value for j */
}
process (int(*pf)(int,int)       /* host function definition */
                                 /* formal argument is a pointer to a function */
{
  int a,b,c;
  ____________
  C = (*pf) (a,b),      /* access the function passed to this function return a
  ______________           value for C */
  return (c);
}
```

```
funct1 (a,b)            /* guest function definition */
int a,b;
  {
      int c;
      C = ________        /* use a and b to evaluate c */
      return (c);
  }
  funct 2 (x,y)           /* guest function definition */
  int x,y;
  {
    int z;
    z = ________          /* use x and y to evaluate z */
    return (z);
  }
```

Note that this program contains 3 function declarations. The declarations of funct 1 and funct 2 are straight forward. However the declaration for process requires some explanation. This declaration states that process is a host function that returns an integer quantity and has one argument. The argument is a pointer to a guest function that returns an integer quantity and has two integer arguments. The argument designation for the guest function is written as

```
int (*) (int, int)
```

Note that way the argument designation fits into the entire host function declaration. (ie)

```
int process (int (*) (int, int));
```

Consider the formal argument declaration that appears within process (ie) int (*pf) (int,int);

This declaration states that pf is a pointer to a guest function. The guest function will return an integer quantity and it requires two integer arguments.

Here is another version of the same outline, utilizing full function prototyping.

```
int process (int (*pf) (int a, intb)); /* function prototype */
int funct1  (inta,intb); /* function prototype guest */
int funct2  (inta, intb); /* function prototype guest */
main ()
{
```

```
int:,i,j;
__________
i = process (funct 1); /* pass funct 1 to process; return a value for i */
j = process (funct 2); /* pass funct 2 to process; return a value for j */
__________
}
process (int (*pf)(inta,intb)) /* host function definition *
{
  int a,b,c;
  ____________
  C = (*pf) (a,b),    /* access the function passed to this function return a
                         value for C */
  ____________
  return (c);
}
funct1 (inta,intb)              /* guest function definition  */
{
   int c;
   C = ________         /* use a and b to evaluate c */
   return (c);
}
  funct 2 (intx,inty)           /* guest function definition */
  {
     int z;
     z = ________               /* use x and y to evaluate z */
     return (z);
  }
```

The function prototypes include argument names as well as argument data types. Moreover, the prototype for process now includes the name of the variable (pf) that points to the guest function. Note that the declaration of the formal argument pf within process is consistent with the function prototype.

Program: Future value of Monthly Deposits.
(Compound interest calculations)

Suppose a person decides to save a fixed amount of money at the end of every month for a year. If the money earns interest at i percent per year, then it is natural to ask how much money will accumulate after n years (ie after 12n monthly deposits). The answer, of course, depends upon how much money is deposited each month, the interest rate, and the frequency with which the interest is compounded.

For example, if the interest is compounded annually, semiannually, quarterly, or monthly, the future amount of money that will accumulate after n years is given by

$$F = \frac{12\,A}{m}\left[\frac{(1+i/m)^{mn}-1}{i/m}\right] = 12\,A\left[\frac{(1+i/m)^{mn}-1}{i}\right]$$

where F is the future accumulation,

A is the amount of money deposited each month,

i is the annual interest rate (expressed as decimal)

& m is the number of compounding periods. Per year.

(eg. m = 1 for annual compounding,

m = 2 for semiannual compounding,

m = 4 for quarterly compounding,

m = 12 for monthly compounding)

If the compounding periods are shorter than the payment periods, such as in the case of daily compounding then the future amount of money is determined by

$$F = A\left[\frac{(1+i/m)^{mn}-1}{(1+i/m)^{m/12}-1}\right]$$

Note that n is customarily assigned a value of 360 when the interest is compounded daily.

Finally, in the case of continuous compounding, the future amount of money is determined as

$$F = A\left[\frac{e^{in}-1}{e^{i/12}-1}\right]$$

The entire program is shown below:

```
/* Personal finance calculations */
# include < stdio.h >
# include < stdib.h >
# include < ctype.h >
# include < math.h >
/* function prototypes */
void table (double (*pf) (doublei, int m, double n),
                 double a, int m, double n);
double md1 (double i; int m, double n);
```

```
double md2 (double i; int m, double n);
double md3 (double i, int m, double n);
main ()   \* calculate the future value of a series of monthly deposits */
{
   int n; \* number of compounding periods per year */
   double n; \* number of years */
   double a; \* amount of each monthly payment */
   char freq; \* frequency of compounding indicator */
   /* enter input data */
   printf ("\n Future value of a series of monthly deposits \n\n");
   printf ("Amount of each monthly payment:");
   Scanf ("%lf", & a);
   printf ("Number of years:");
   Scanf ("%lf", & n);
   /* enter frequency of compounding */
   do
      {
      printf ("frequency of compounding (A,S,Q,M,D,C):");
      Scanf ("%ls", & freq);
      Freq = toupper (freq); \* convert to upper case */
      if  (freq==`A')
      {
          m = 1;
          printf ("\n Annual compounding \n");
      }
      else if (freq == `S')
      {
          m = 2;
          printf ("\n Semiannual compounding \n");
     }
     else if (freq = = `Q')
     {
          m = 4;
        printf ("\n Quarterly compounding \n");
     }
     else if (freq == `M')
     {
        m = 12;
        printf ("\n monthly compounding \n);
    }
    else if (freq == `D')
    {
```

```
        m = 360;
        printf ("\n Daily compounding \n");
    }
    else if (freq == `C')
    {
        m = 0;
        printf ("\n continuous compounding \n");
    }
    else
        printf ("\n ERROR – Please Repeat \n\n");
   {
        while (freq ! = `A' && freq ! = `S' &&
                freq ! = `Q' && freq ! = `M' && freq ! = `D'
                && freq ! = `C');
        /* carry out the calculations */
        if (freq == `C')
                table (md 3, a, m, n); /* continuous compounding */
        else if (freq == `D')
                table (md 2, a,m,n); /* daily compounding */
        else
                table (md 1, a, m,n); /* annual, semi annual, quarterly or
                                                  monthly compounding */
   }
   void table (double (*pf) (double i, intm, double n),
                   double a, intm, double n)
   /* table generator (this function accepts a pointer to another
        function as an argument)
   Note: double (*pf) (double i, intn, double n)
          Is a POINTER TO A FUNCTION */
{
   int count; /* loop counter */
   double i;  /* annual interest rate */
   double f; /* future value */
   printf ("\n Interest Rate Future Amount \n\n");
   for  (count = 1; count < = 18;  ++ count)
   {
        i = 0.01 * count;
        f = a * (*pf) (i,m,n);  /* Access the function passed as a pointer */
        printf ("%2d %2f\n", count, f);
   }
   return;
}
```

```
double md1 (doublei, intn, double n)
/* ,monthly deposits, periodic compounding */
{
   double factor, ratio;
   factor = 1 + i/m;
   ratio = 12 * (pow (factor, m * n) – 1)/i;
   return (ratio)
}

double md2 (doublei, intm, double n)
/* monthly deposits, daily compounding */
{
   double factor, ratio;
   factor = 1 + i/m;
   ratio = (Pow (factor, m * n) – 1) / (pow (factor, m/12)–1);
   return (ratio);
}
double md3 (doublei, int dummy, double n)
/* monthly deposits, continuous compounding */
{
   double ratio;
   ratio = (exp (i*n)–1) / (exp (i/12)–1);
   return (ratio)
}
```

Notice the function prototypes, particularly the prototype for table. The first argument passed to table is a pointer to a guest function that receives two double precision arguments and an integer argument, and returns a double precision quantity. This pointer is intended to represent md1, md2 or md3. The prototypes of these three functions follow the prototype for table. Each of these functions accepts two double precision arguments and an integer argument, and returns a double precision quantity as required.

```
Future value of a series of Monthly Deposits

Amount of each monthly payment: 100
Number of years: 3
Frequency of compounding (A,S,Q,M,D,C):h
ERROR – Please Repeat
Frequency of compounding (A,S,Q,M,D,C): m
Monthly compounding
```

Interest Rate	Future Amount
1	3653.00
2	3707.01
3	3762.06
4	3818.16
5	3875.33
6	3933.61
7	3993.01
8	4053.56
9	4115.27
10	4178.18
11	4242.31
12	4307.68
13	4374.33
14	4442.28
15	4511.55
16	4582.17
17	4654.18
18	4727.60

6.8. More About Pointer Declarations:

Pointer declarations can become complicated, and some care is required in their interpretation. This is especially true of declarations that involve functions or arrays.

One difficulty is the dual use of parentheses. In particular, parentheses are used to indicate functions and they are used for nesting purposes within more complicated declarations.

Several declarations involving pointers are shown below:

```
int P;               /* P is a pointer to an integer quantity */
int *P [10];         /* P is a 10-element array of pointers to integer quantities */
int (*P) [10];       /* P is a pointer to a 10-element integer array */.
int *P (void);       /* P is a function that returns a pointer to an integer quantity
                     */
int P (char *a);     /* P is a function that accepts an argument which is a pointer
                     to a character and returns an integer quantity */
int *P (char *a);    /* P is a function that accepts an argument which is a pointer
                     to a character returns a pointer to an integer quantity */
itn (*P) (char *a);  /* P is a pointer to a function that accepts an argument which
                     is a pointer to a character returns an integer quantity */
```

```
int (*P (char *a)
          [10];      /* P is a function that accepts an argument which is a pointer
                     to a character returns a pointer to a 10-element integer array
                     */
int P (char (*a) [ ]); /* P is a function that accepts an argument which is a pointer
                     to a character array returns an integer quantity */
int P (char *a [ ]);   /* P is a function that accepts an argument which is an array
                     of pointers to characters returns an integer quantity */
int *P (char a [ ]);   /* P is a function that accepts an argument which is a
                     character array returns a pointer to an integer quantity */
int *P (char (*a) [ ]); /* P is a function that accepts an argument which is a pointer
                     to a character array returns a pointer to an integer quantity */
int *P (char *a [ ]);  /* P is a function that accepts an argument which is an array
                     of pointers to characters returns a pointer to an integer
                     quantity /*
int (*P) (char)
      (*a) [ ]);     /* P is a pointer to a function that accepts an argument which
                     is a pointer to a character array returns an integer quantity */
int * (*P)
     (char (*a) [ ]); /* P is pointer to a function that accepts an argument which is
                     a pointer to a character array returns a pointer to an integer
                     quantity */
int * (*P)
    (char *a [ ]);    /* P is a pointer to a function that accepts an argument which
                     is an array of pointers to characters returns a pointer to an
                     integer quantity */
int (*P [10]) (void); /* P is a 10 – element array of pointers to functions; each
                     function returns an integer quantity */
int (* P [10] (char a);/* P is a 10 – element array of pointers to functions; each
                     function accepts an argument which is a character, and
                     returns an integer quantity */
int * (*P [10])
      (char a);      /* P is a 10-element array of pointers to functions; each
                     function accepts an argument which is a character and
                     returns a pointer to an integer quantity */

int * (*P [10]
(char * a);          /* P is a 10-element array of pointers to functions; each
                     function accepts an argument which is a pointer to a
                     character and returns a pointer to an integer quantity */
```

UNIT – 7
STRUCTURES AND UNIONS

The C language gives you five ways to create custom data types:

1. The structure is a grouping of variables under one name and sometimes called a conglomerate data type.
2. The bit field is a variation of the structure and allows easy access to the individual bits within a word.
3. The union enables the same piece of memory to be defined as two or more different types of variables.
4. The enumeration is a list of symbols.
5. The typedef keyword simply creates a new name for an existing type.

7.1: Structures:

In C, a structure is a collection of variables that are referenced under one name, providing a convenient means of keeping related information together. In general terms, the composition of a structure may be defined as

```
Struct tag
{
     member 1;
     member 2;
     -------------
     member m;
};
```

in this declaration, Struct is a required keyword; tag is a name that identifies structures of this type (ie, structures having this composition); and member1 through member m are individual member declarations. The individual members can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, through a member name can be the same as the name of the variable that is defined outside the structure. A storage class however cannot be assigned to an individual member, and individual members cannot be initialized within a structure type declaration.

Points to be noted:

1. The composition of the structure is terminated with a semicolon.
2. While the entire declaration is considered as a statement, each member is declared independently for its name and type in a separate statement inside the composition.

3. The tag name can be used to declare structure variables of its type later in the program.

Once the composition of the structure has been defined, individual structure type variables can be declared as follows.

Storage– class struct tag var1, var2, -----varn; where storage–class is an optional class specifier, struct is a required keyword, tag is the name that appeared in the structure declaration, and var1, var2, ----- varn are structure variables of type tag.

A typical structure declaration is shown here:

```
Struct account
{
int accno;
char acctype;
char name[80];
float balance;
};
```

This structure is named account (ie. the tag is account) It contains four members; an integer quantity (accno), single character (acc type) an 80 element character array(name[80]), and a floating point quantity(balance).

We can now declare the structure variables oldcustomer and newcustomer as follows:

Struct account oldcustomer, newcustomer; Thus, old customer and newcustomer are variables of type account. In other words, oldcustomer and newcustomer are structure type variables whose composition is identified by the tag account.

It is possible to combine the declaration of the structure composition will that of the structure variables as shown below:

```
Struct account
{
    int accno;
    char acctype;
    char name[80];
    float balance;
} oldcustomer, newcustomer;
```

The tag ie. account is optional in this case.

7.2: Initializing Structures:

The members of a structure variable can be assigned intial values in much the same manner as the elements of an array. The intial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

Storage–class struct tag var = {val1, val2, ………, valn} where val1 refers to the first member val2 refers to the second member, and so on. A structure variable, like an array, can be initialized only if its storage class is either external or static.

```
Struct account
{
     int accno;
     char acctype;
     char name[80];
     float balance;
};
```

```
Static struct account customer: {12334, 'R', "JANEE", 2345.80};
```

Thus customer is a static structure variable of type account, whose members are assigned intial values. The first member is assigned the integer value 12334, the second member is assigned the character 'R', the third member is assigned the string "JANEE", and the fourth member is assigned the floating – point value 2345.80.

Referencing Structure Members:

Individual structure members are referenced by using the (dot) operator. For example the following code assigns the value 12335 to the accno of the structure variable customer declared earlier:

```
Customer. Accno = 12335;
```

The structure variable name followed by a period and the member name references that individual member. All structure members are accessed in the same manner.

The general form is

```
Structure – name . member - name
```

Therefore, to print the accno to the screen, you could write

```
Printf("% ld", customer . accno);
```

This prints the accno contained in the accno variable of the structure variable customer.

Program:
Define and assign values to structure members:

```
# include<stdio.h>
struct personal
{
      char name[20];
      int day;
      char month[10];
      int year;
      float salary;
};
main( )
{
    struct personal person;
    printf("Input values \n");
    scanf("%s %d %s %d %f", person.name, &person.day,
                  person.month, & person.year, & person.salary);
    printf("%s %d %s %d %f", person.name, person. day,
                                  person. month, person.year);
}
```

Output:

```
Input values
Wallis 26   June   2001   6800
Wallis  26   June   2001   6800
```

Program:
To initialise members in a structure:

```
# include<stdio.h>
# include<string.h>
main( )
{
      struct emp
      {
         char name[20];
         char sex;
```

```
      int empno;
    };
    struct emp empinfo;
    strcpy (empinfo.name, "John");
    empinfo.sex ='M';
    empinfo. empno = 3023;
    printf(" \n Employee Name : %s", empinfo.name);
    printf("\n sex: %c", empinfo.sex);
    printf(\n sex: %c", empinfo.sex);
    printf(\n Employee No : %d", empinfo. empno);
}
```

Output:

```
Employee Name : John
Sex : M
Employee No : 3023
```

Comparison of structure variables:

Two variables of the same structure type can be compared the same as the ordinary variables. If person1 and person2 belong to the same structure ᵗʰ . the following operations are valid:

Person1 = person2	assign person2 to person1
Person1 == person2	compare all members of person1 and Return 1 if equal, 0 otherwise
person1 ! = person2	return 1 if all members are not equal, 0 0therwise.

The program shown below illustrates how a structure variable can be copied into another of the same type. It also performs member–wise comparison to decide whether two structure variables are identical.

Program:

Comparison of structure variables:

```
# include<stdio.h>
struct class
{
    int num;
    char name[20];
    float marks;
};
main( )
```

```
{
    int x;
    static struct class student1 = { 101, "CATHY", 80.5};
    static struct class student2 = {105, "ROVINA", 78.5};
    struct class student3;
    student3 = student2;
    x = ((student3. num == student2.num) &&
                        student3. marks == student2.num))?1:0;
    if (x==1)
    {
        printf("\n student2 and student3 are same \n \n");
        printf("%d %s %f \n",  student3.num,
                              student3.name, student3.marks);
    }
    else
    printf("\n student2 and student3 are different");
}
```

Output:

```
Student2 and student3 are same
 105 ROVINA 78.500000
```

Arrays of structures:

Perhaps the most common use of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare are array variable of that type. For example, to declare a 100 – element arrays of structure of type account, which was declared earlier in this chapter, you would write

```
struct account customer[100];
```

This creates 100 sets of variables that are organized as declared in the structure type account.

To access a specific structure within the customer array, the structure variable name is indexed. For example, to print the balance of the third structure, you would write

```
printf("%d", customer[2].balance);
```

Like all array variables, arrays of structures begin their indexing at o.

To help illustrate how structures and arrays of structures are used, consider a simple program to the subject wise and student total and store them as part of the structure.

Program: array of a structure.

```
# include<stdio.h>
struct marks
{
    int sub1;
    int sub2;
    int sub3;
};
main( )
{
    int I;
    static struct marks student[3] = {{47, 67, 81,0},
                                      {75,54,69,0},
                                      {57,36,71,0}};
    static struct marks total;
    for (i=0; i<=3; i++)
    {
        student [i]. total = student [i]. sub1 + student [i].sub2 +
                                                  student [i]. sub3;
        total.sub1 = total.sub1 + student [i]. sub1;
        total.sub2 = total.sub2 + student [i].sub2;
        total.sub3 = total.sub3 + student [i].sub3;
        total.total =total.sub1 + student [i].total;
    }
        for (i=0; i<=2; i++)
        printf("student [%d] %d \n", i+1, student[i].total);
        printf(" \n SUBJECT                TOTAL \n \n");
        printf(" %s            %d \n  %s %d \n %s %d \n"),
               "subject 1", total.sub1,
               "subject2", total.sub2,
               "subject3", total.sub3);
        printf(" \n Grand total = %d \n", total.total);
    }
```

Program:

```
# include<stdio.h>
main( )
{
    struct emp
    {
        char name[20];
        int empno, bp;
    };
    struct emp empinfo[10];
    int n,I;
    printf("enter how many employees are there \n");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        printf("enter employee name \n");
        scanf("%s", empinfo[i].name);
        printf("enter employee number \n");
        scanf("%d", & empinfo[i].empno);
        printf("enter basic pay \n");
        scanf("%d", &emp info[i].6p);
    }
    printf(" \n output: \n");
    for (i=0; i<n; i++)
    {
        printf(" \n Employee Name : %s,"empinfo [i].name);
        printf("\n Employee No: %d", empinfo[i].empno);
        printf("\n Basic pay: %d", empinfo [i].6);
    }
}
```

Output:

```
enter how many employees are there
2
enter employee name
Saro
enter employee number
5
enter basic pay
3000
enter employee name
Helen
```

```
enter employee Number
10
enter basic pay
4000
```

Output:

```
Employee Name    :  Saro
Employee No      :  5
Basic pay        :  3000
Employee Name    :  Helen
Employee No      :  10
Basic pay        :  4000
```

Program:

Program to initialize some members of an array of structures and to display the contents of all the structures.

```
# define MAX 5
# include<stdio.h>
main()
{
    struct student
    {
        long int rollno;
        int age;
        char sex;
        float height;
        float weight;
    };
    static struct student class [MAX] ={
                                          {9101, 25, 'M', 178.3, 72.1},
                                          {9102, 24, 'F', 164.3, 58.2},
                                          {9103, 24, 'M', 180.2, 80.8},
                                          {9104,25, 'F', 167.8, 40.0}
                                          };
    int i;
    printf("contents of structure \n");
    for(i=0; i<=MAX-1; i++)
    {
        printf("RollNo : %ld \n", class[i]. rollno);
        printf("Age : %d \n", class[i].age);
        printf("sex: %c \n", class[i].sex);
        printf("weight:%0.2f \n", classs[i].weight);
    }
}
```

Arrays within structures:

C permits the use of arrays as structure members. We have already used arrays of characters inside the structure. Similarly we can use single or multidimensional arrays of type int or float. For example, the following structure declaration is valid:

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

here the member subject contains three elements, subject[0], subject[1] and subject[2].
These elements can be accessed using appropriate subscripts.

For example, the name
    Student[1]. Subject[2];
would refer to the marks obtained in the 3rd subject by the 2nd student.

Program:

A program to read a set of names of the students from the keyboard and to sort them in an ascending order using a structure within a array data type.

```
# include<stdio.h>
# include<string.h>
main()
{
    struct student
        {
            char name[20];
        };
    struct student a[100];
    input();
}
input()
{
    struct student a[100];
    int i,n,j,ch;
    printf("How many names ? \n");
    scanf("%d", &n);
    getchar();                              /* line check */
```

```
   for(i=0; i<=n–1; ++;)
   {
       printf("Name:");
       j=0;
       do
           {
               ch = getchar();
               a[i].name[j] = ch;
               j++;
           }
        while (ch! = '\n');
        a[i].name[j++] = '\o';
   }
   printf("\n\n");
   printf("unsorted form :: \n");
   output (a,n);
   printf("\n \n sorted form :: \n");
   sort(a,n);
   output (a,n);
}

output (a,n)
struct student a[100];
int n;
{
    int i,j,ch;
    printf("\n\n");
    for(i=0; i<=n–1; ++i)
    {
        j=0;
        do
        {
            ch=a[i].name[j++];
            putchar (ch);
        }
        while (ch! = '\n');
    }
 }
 sort(a,n)
 struct student a[100];
 int n;
 {
```

```
        int i,j,ch;
        char temp[20];
        for(i=0; i<=n -1; ++i)
        {
            for(j=0; j<=n -2; ++j)
        if(strcmp(a[i].name, a[j].name) <=0)
        {
            strcpy(temp, a[i].name);
            strcpy(a[i].name, a[j].name);
            strcpy(a[j].name, temp);
        }
    }
}
```

Output:

```
How many names?
3
Name : Janne
Name : George
Name : Johnny

    Unsorted form ::
Janee
George
Johnny
    Sorted form ::
George
Janee
Johnny
```

Program:

A program to read a set of names, rollno, sex, height, weight of the students from the keyboard and to sort them in an ascending order using a structure within an array data type.

```
# include<stdio.h>
# include<string.h>
main()
{
    struct student
    {
        char name[20];
```

```
        long int rollno;
        char sex;
        float height;
        float weight;
    };
    struct student a[50];
    input( );
}
    struct student a[50];
    int i,n,j,ch;
    printf("How many names? \n");
    scanf("%d", &n);
    getchar( );                          /* line check */
    for(i=0; i<n–1; ++i)
    {
        printf("Name : ");
        j = 0;
        do
            {
                ch=getchar( );
                a[i].name [j] =ch;
                j++;
            }
            while (ch! = '\n');
            a[i].name [j++] = ' \o ';
            printf("Roll No : ");
            scanf("%d", &a[i].rollno);
            printf("sex :");
            scanf("%2s", &a[i].sex);
            printf("height");
            scanf("%f", &a[i].height);
            printf("weight");
            scanf("%f", &a[i].weight);
            getchar();
            printf("\n\n");
        printf("Unsorted form : : \n");
        output(a,n);
        printf("\n\n sorted form : :\n");
        sort(a,n);
        output(a,n);
    }
    output(a,n)
```

```
    struct student a[50];
    int n;
    {
        int i,j,ch;
        printf("\n\n");
        printf("Name    RollNo    Sex    Height    Weight \n");
        printf(" ------------------------------------------------- \n");
        for(i=0; i<n–1; ++i)
        {
            j = 0;
            ch = '\o';
            while (ch! = '\n')
            {
                putchar (ch);
                ch=a[i]. name [j++];
            }
        printf(" \t % 15ld", a[i]. rollno);
        printf("\t %5s", a[i].sex);
        printf(" \t %15.2f", a[i].height);
        printf("\t %15.2f", a[i].weight);
        printf("\n");
    }
}
sort (a,n)
struct student a[50];
int n;
{
    struct student temp;
    int i,j,ch;
    for (i=0; i<=n–1; ++i)
    {
        for j=0; j<=n–2; ++j)
        if (strcmp (a[i].name, a[j].name)<=0)]
        {
            temp =a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
```

Output:

How many names?

4

Name : jesu
Roll No : 8901
Sex : M
Height : 167
Weight : 67

Name : antony
Roll no : 8902
Sex : M
Height : 189
Weight : 78

Name : rajam
Roll No : 8903
Sex : F
Height : 172
Weight : 71

Name : Sophia
Roll No : 8904
Sex : F
Height : 164
Weight : 55

Unsorted form : :

Name	RollNo	Sex	Height	Weight
jesu	8901	M	167	67
antony	8902	M	189	78
rajam	8903	F	172	71
Sophia	8904	F	164	55

Sorted form : :

Name	RollNo	Sex	Height	Weight
antony	8902	M	189	78
jesu	8901	M	167	67
rajam	8903	F	172	71

Sophia 8904 F 164 55

Structures within structures: (Nested structure)

A structure may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

For example,

```
struct date
{
    int month;
    int day;
    int year;
};
struct account
{
    int accno;
    char acctype;
    char name[80];
    float balance;
    struct date last payment;
} old customer, new customer;
```

The following code fragment assigns 1990 to the year element of date and 12330 to the accno member of account:

```
Account. accno = 12330;
Account.date.year = 1990;
```

The elements of each structure are referenced from outermost to innermost (left to right). Structures can be nested up to any level provided there is sufficient memory.

Program:

```
main()
{
    struct mark
    {
        int inter, exter;
    };
    struct stdrec
    {
        char name[20];
```

```
        int regno;
        struct mark markrec;
    };
    struct stdrec exam;
    int tot;
    printf ("enter student name \n");
    scanf("%s", exam.name);
    printf("exer examno \n");
    scanf("%d", &exam.regno);
    printf("internal mark \n");
    scanf("%d", &exam. Markrec. inter\n");
    printf("external mark \n");
    scanf("%d", &exam.markrec.exter);
    tot=exam.markrec.inter +exam.markrec.exter;
    printf("\n regno: %d" exam.regno);
    printf("\n total : %d", tot);
```

Output:

```
enter student name
kishore
enter examno
5
internal mark
22
external mark
67
name : kishore
regno : 5
total  : 89
```

In this case markrec is the inner structure of the structure stdrec. Member elements can be accessed by linking all the structure variable (from. outer structure to in the inner structure) that are connected with the member using dot operator.

User – defined data type: typedef

The typedef feature allows users to define new data types that are equivalent to existing data types. Once a user – defined data type has been established, then new variables, arrays, structures etc can be declared in terms of this new datatype.

In general terms, a new data type is defined as

```
typedef  type new –type ;
```

where type refers to an existing data type and new–type refers to the new user – defined data type. It should be understood that the new data type would be new in name only. In reality, this new data type will not be fundamentally different from one of the standard datatypes.

Here is a simple declaration of the use of typedef:

```
typedef int number;
```

In this declaration, number is a new name given to the existing datatype int. Hence the variable declaration.

```
number count, age;
```

is equivalent to writing

```
int count, age;
```

In other words, count and age are treated as of type number, although they are actually integer – type variables.

The typedef feature is particularly convient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referred. Hence the structure can be referred more concisely. In addition, the name given to a user – defined structure type often suggests the purpose of the structure within the program.

In general terms, a user – defined structure type can be written as:

```
typedef struct
    {
        member 1;
        member 2;
        -------------
        member n;
    } new – type;
```

where newtype is the user – defined structure type. Structure variables can be then defined in terms of the new data type.

Consider the following declaration:

```
typedef struct
{
    int accno;
    char acctype;
    char name[80];
    float balance;
} record;
```

```
record oldcustomer, newcustomer;
```

The first declaration defines as a user – defined data type. The second declaration defines old customer and newcustomer as structure variables of type record.

Program:

A program to define the variables using typedef and to display the contents of the variable

```
/* using typedef */
# include<stdio.h>
main()
{
    typedef int integer;
    typedef char character;
    typedef float real;
    integer i,j;
    character ch;
    real a,b;
    i = 10;
    j = 30;
    ch = 'm';
    a =–12.45;
    b= 34.67;
    printf("using typedef \n");
    printf(" %d \t %d \t %2c \t % 0.2f \t % 0.2f\n", I,j,ch,a,b);
}
```

Output of the above program:

Using typedef
10 30 M – 12.45 4.67

Program:

A program to declare the member of a structure using typedef and to display the contents of the structure.

```
#include<stdio.h>
main()
{
    struct first
    {
        int a;
        float b;
```

```
        charc;
    };
    typedef struct first number;
    number one;
    one.a = 23;
    one.b = 11.89;
    one.c = 'f';
    printf("%d \t %0.2f \t %0.2c \n", one.a, one.b, one.c);
}
```

Output:

```
23      11.89       f
```

The following typedef declaration in the structure data type is valid:

1. typedef struct first

```
{
int a;
float b;
} number;
number one;
```

2. typedef struct

```
{
int day;
float month;
char year;
} date;
typedef struct
{
    char name[20];
    int rollno;
date dob;
} student;
student a[200];
```

3. typedef struct

```
{
int day;
int month;
int year;
} date;
typedef struct
```

```
{
    char name [20];
    int rollno;
    date dob;
} student [200];
student a;
```

4. typedef struct

```
{
    int day;
    int month;
    int year;
} date;
struct
{
    char name[20];
    int rollno;
    date dob;
} a[200];
```

Structures and Functions:

So far, all structures and arrays of structures used in the examples have been assumed to be either global or defined within the function that uses them. Here special consideration will be given to passing structures and their elements to functions.

(a) Passing structure elements to functions:

When you pass an element of a structure variable to a function, you are actually passing the value of that element to the function. Therefore, you are passing a simple variable.

For example, consider this structure

```
struct example
{
    char c;
    float f;
    int i;
    char s[10];
} test;
```

Here are examples of each element being passed to a function:

```
func (example.c);             /* Passes character value of c*/
```

```
func (example.f);           /* passes float value of f */
func (example.i);           /* passes integer value of i */
func (example.s);           /* passes character value of s */
func (example. S[2]);       /* passes character value of s[2] */
```

Howerver, if you wished to pass the address of individual structure elements to achieve call by reference parameter passing, you would place the & operator before the structure name.

For example, to pass the address of the elements in the structure example, you would write

```
func (& example.c);        /* passes address of char c */
func (& example.f);        /* passes address of float f */
func (& example.i);        /* passes address of int i */
func (& example.s);        /* passes address of strings */
func (&example.s[2]);      /* passes address of s[2] */
```

Notice that the & operator preceder the structure name, but not the individual element name. Note also that the string element s already signifies an address, so that no & is required. However, when accessing a specific character in string s, as shown in the final example, the & is still needed.

(b) Passing Entire structures to functions:

When a structure is used as an argument to a function, the entire structure is passed using the standard call by method. This means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

When using a structure as a parameter, the most important thing to remember is that the type of the argument must match the type of the parameter. The best way to do this is to define a structure globally and then use its name to declare structure variables and parameters as needed.

For example:

```
# include<stdio.h>
struct struct_type
{
    int a,b;
    char ch;
};
void func(struct struct_type parm);
```

```
main()
{
    struct struct_type arg;
    arg.a = 1000;
    func(arg);
}
func (struct  struct– type parm)
{
    printf("%d", parm.a);
}
```

This program prints the number 1000 on the screen. As you can see, both arg and parm are declared to be structures of type struct–type.

It is important to observe while passing structures to functions that;

(a) the called function must be appropriately declared for its type, data type and struct tag – name.

(b) The structure element and the corresponding argument in the called function must be of same struct type.

(c) The use of return statements depend on how the situation warrants for it.

(d) When a function returns a structure, it must be assigned to a structure of identical type in the calling function.

(e) The called function must be declared in the calling function for its type, if it is placed after the calling function.

7.3: Structures and Pointers:

So far, it has been seen that a member of a structure could be an ordinary data type such as int, float, char and even a structure also. In c, it is permitted to declare a pointer variable as a member to a structure. It is well known that a pointer is a variable which holds the memory address of a variable of any basic data types such as int, float or sometimes an array. A pointer can be used to hold the address of a structure variable too. The pointer variables is very much used to construct a complex data base using the data structures such as linked lists, double linked lists, binary trees etc.

The following declaration is valid

For example

```
struct sample
{
    int x;
    float y;
    char s;
}
struct sample *ptr;
```

wher ptr is a pointer variable holding the address of the structure sample and is having the 3 members such as int x, float y, char s. The pointer structure variable will be accessed an processes in one of the following ways.

(* structure name). field name = variable;

The parentheses are required, since the structure member period ( ) has higher precedence than the indirection operator (*), or the pointer structure can be expressed using (–) followed by the greater than (>).

Structure name → field name = variable;

The following assignment is valid using pointer structure

Case –1:

```
main()
{
    struct sample
        {
            int x;
            float y;
            char s;
        }
struct sample *ptr;
(*ptr).x =10;
(*ptr).y = – 23.4;
(*ptr).s = 'j';
---------------------
---------------------
}
```

case 2:

```
main()
{
    struct sample
    {
        int x;
        float y;
        char s;
    };
    struct sample * ptr;
    ptr → x = 10;
    ptr → y = – 23.45;
```

```
    ptr → s = ' j ';
    -------------------------
    -------------------------
}
```

Program:

A program to assign some values to the member of the structure using an in direction operator.

```
#include<stdio.h>
main()
{
    struct sample
        {
            int x;
            int y;
        };
        struct sample one;
        struct one *ptr;
        ptr =&one;
        (*ptr).x = 10;
        (*ptr).y = 20;
        printf("contents of x = %d \n", (*ptr).x);
        printf("contents of y = %d\n", (*ptr).y);
}
```

Output:

```
Contents of x =10
Contents of y = 20
```

Program:

A program to assign some values to the member of the structure using a pointer structure operator.

```
#include<stdio.h>
main()
{
    struct sample
    {
        int x;
        int y;
    };
    struct sample one;
```

```
        struct one *ptr;
        ptr = &one;
        ptr → x=10;
        ptr → y=20;
        printf("contents of x=%d \n", ptr → x);
        printf("contents of y=%d \n", ptr → y);
    }
```

Output:

```
    Contents of x = 10
    Contents of y = 20
```

Program:

A program to read a set of values from the keyboard using a pointer structure operator and to display the contents of the structure onto the screen.

```
    #include<stdio.h>
    main()
    {
        struct sample
        {
            int x;
            int y;
        };
        struct sample one;
        struct one *ptr;
        printf("Enter value for x and y? \n");
        scanf("%d %d", &ptr → x, &ptr → y);
        printf("contents of x and y \n");
        printf("%d %d      \n", ptr→ x, ptr→ x);
    }
```

Output:

```
    Enter value for x and y?
    10 20
    contents of x and y
    10 20
```

Pointer variable as a member of a structure:

A pointer can also be used as member of a structure. The following structure declaration is valid.

```
struct sample
        {
            int *ptr1;
            float *ptr2;
        };
        struct sample *first;
```

Program:

A program to declare a pointer variable as a member of a structure and to display the contents of the structure.

```
#include<stdio.h>
main()
{
    struct sample
    {
        int *ptr1;
        float *ptr2;
    };
struct sample *first;
int value1;
float value2;
value1 = 10;
value2 = 20.01;
first → ptr1 = & value1;
first → ptr2 = & value2;
printf("contents of the first member = %d \n", *first → ptr1);
printf("contents of the second member = %0.2f \n", *first → ptr2);
}
```

Output:

```
Contents of the first member = 10
Contents of the second member = 20.01
```

Unions:

In C, union is a memory location that is used by several variables, which are of different types. The union declaration is similar to that of a structure as shown in this example.

```
Union union – type
{
    int i;
    char ch;
};
```

As with structures, you may declare a variable either by placing its name at the end of the definition or by using a separate declaration statement. To declare a union variable cnvt of type union – type using the definition just given, you would write

```
Union union – type cnvt;
```

In union cnvt, both integer; and character ch share the same memory location.

When a union is declared, the compiler automatically creates a variable large enough to hold the largest variable type in the union. To access a union element use the same syntax that you would use for structures. If you are operating on the union directly, use the dot operator. For example to assign the integer 10 to element I of cnvt, you would write

```
Cnvt.i = 10;
```

Unions are frequently used when type conversions are needed. For example, the standard library function putw() writes the binary representation of an integer to a disk file. First a union composed of one integer and a 2 – byte character array is created:

```
Union pw
{
    int i;
    char ch[2];
};
```

now, putw() is written using the union

Program:-

A program to initialize the member of a union and to display the contents of the union.

```
#include<stdio>
main()
{
    union value
    {
     int i;
     float f;
```

```
        };
 union value x;
        x.i = 10;
        x.f = –1456.45;
 printf("First member %d\n", x.i);
 printf("Second member %0.2f\n",x.f);
 }
```

Output:-

First member 3686
Second member – 1456.45

In the above program, the union is declare to consists two members such as an int and a float. Only the float values are stored and displayed correctly and the integer values are displayed wrongly. The union only holds a value for one data type of the larger storage of their members.

Program:-

A program to declare a member of a union as a structure data type and to display the contents of the union.

```
#include<stdio.h>
main()
{
struct date
{
int day;
int month;
int year;
};
union value
{
int i;
float f;
struct date bdate;
};
union value x;
x.i=10;
x.f= – 1456.45;
x.bdate.day = 12;
x.bdate.month =7;
x.bdate.year = 1992;
```

```
printf("First member %d\n", x.i);
printf("Second member%0.2f\n",x.f);
printf("Structure:\n");
printf("%d / %d / %d\n",x.bdate.day,x.bdate.month, x.bdate.year);
}
```

Output:-

```
First member 12
Second member 0.00
Structure:
12/7/1992
```

Program:-

A program to declare a union as a pointer data type and to display the contents of the union using pointer operator.

```
#include<Stdio.h>
main()
{
    union value
    {
    int i;
    float f;
};
union value *a; /*pointer declaration*\
  a → i = 10;
  a → f =–1456.45;
  printf("First member %d \n", a →i );
  printf("Second member %0.2f \n", a → f);
}
```

Output:

```
First member 3686
Second member – 1456.45
```

7.4: Bit fields:

A bit field is a special type of structure member, in that several bit fields can be packed into an int. while bit fields are variables, they are defined in terms of bit rather than characters or integers. Bit fields are useful for maintaining single or multiple bit flags in an int without having to use logical AND and OR operations to set and clear them. They can also assist with combining and dissecting bytes and words that one sends to and receives from external devices.

The formal declaration of the bit field is same as the declaration of a structure, but there is a difference in accessing and using a bit field in a structure. The number of bits required to a variable must be specified and followed by a colon while declaring a bit field. The bit fields must be signed or unsigned integers from 1 to 16 bits in length. Number of bits will be depending upon the machine you are using.

The bit field is very much useful with data items where only a few bits are required to indicate a true or false condition. Secondly, the bit field is used to save a memory space, as the number of bits required are declared for each variable in a structure. So C will accommodate all these bits into a packed binary form.

The general format of the bit field declaration is as follows:

```
struct user – defined name
{
  member1;
  member2;
  ------------
  member;
};
```

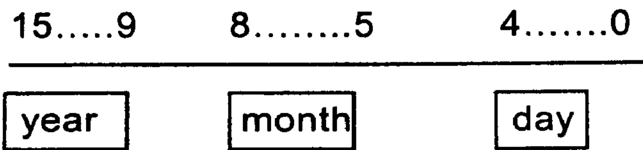
where the individual elements hare the same meaning as in the structure declaration, each member declaration must now indicate a specification indicating the size of the corresponding bit field. To do so, the member name must be followed by a colon and an unsigned integer indicating its size of field. The interpretation of these bit fields may vary from. One C compiler to another. For example, some C compilers may order the bit field from rights to left, whereas other C compilers will order them from left to right.

For example

```
struct date
{
    unsigned int day: 5; /* day is 5 bits*/
    unsigned int month: 4; /* month is 4 bits */
    unsigned int year: 7; /* year is 7 bits */
 };
```

Declare a structure with these fields:

Day, month, and year as shown below



The entire structure bits is a single 16bit word Day takes up 5 bits, month takes up 4 bits and year takes up 7 bits.

The way of accessing a bit field in a structure is similar to accessing struct date birthday;

```
birthday.day = 17;
birthday.month =5;
birthday.year=1967;
```

Here is one restriction, one cannot take the address of a bit field which means that he cannot use scanf to read values into a bit field. Instead, one will have to read into a temporary variable and then assign its value to the bit field.

Even the bit fields may be accessed in a structure using a pointer operator.

For example,

```
struct date
  {
    unsigned int day :5;
    unsigned int month:4;
    unsigned int year:7;
  }:
struct date *bday;
    bday → day = 17;
    bday → day = 5;
    bday → year = 1967;
```

Program:

A program to declare the member of a structure using a bit field datatype and to display the contents of the structure

```
# include<stdio.h>
main()
{
    struct value
    {
        unsigned day :5;
```

```
        unsigned month :4;
        unsigned year; 7;
    };
    struct value a;
        a.day = 12;
        a.month =7;
        a.year=1992;
        printf("Date:%d / %d / %d \n", a.day, a.month, a.year);
        printf("a requires %d bytes \n", size of (a));
  }
```

Output:

```
Date : 12/7/1992
a requires 2 bytes
```

Program:

A program to declare the member of a structure as a bitfield **data type** using a macro definition and to display the contents of the structure.

```
#define BF1 5
#define BF2 4
#define BF3 7

# include<stdio.h>
main()
{
    struct value
    {
        unsigned day : BF1;
        unsigned month : BF2;
        unsigned year : BF3;
    }a;
    a.day = 23;
    a.month = 7;
  a.year=1992;
  printf("Bit field using the macro definitions \n");
  printf("Date: %d / %d / %d \n", a.day, a.month, a.year);
}
```

Output:

```
Bit field using macro definitions
Date: 23/7/1992
```

Program:

A program to initialize the member of a structure as a bit field data type using a macro definition and to display the contents of the structure.

```
#define BF1 5
#define BF2 4
#define BF3 7

# include<stdio.h>
main()
{
    struct value
    {
        int i;
        unsigned day : BF1;
        unsigned month : BF2;
        unsigned year : BF3;
        float f;
    };
    struct value a={10,2,3,7, 1992, -123.4}
    printf("Bit field initialization \n");
    printf("Integer value %d \n", a.i);
    printf("Date: %d / %d / %d \n", a.day, a.month, a.year);
    printf("Floating point value = %0.2 f \n", a.f);
}
```

Output:

```
Bit field intializtion
Integer value 10
Date : 23/7/1992
Floating point value =– 123.40
```

7.5: Enumerations:

An enumeration is a set of named integer constants that specifies all the legal values that a variable of its type can have.

For example, an enumeration of the coins used in united states is
Penny, nickel, dime, Quarter, half – dollar, dollar

Enumerations are defined like structures, by using the keyword enum to signal the start of the enumeration type.

The general form is

enum   enum – tag – name {enumeration – constants}
variable – list;

Both the enumeration name enum – tag – name and variable – list are optional, but one of them must be present. As with structures, the enumeration tag name is used to declare variables of its type. The following fragment defines an enumeration called coin and declares money to be of that type:

```
Enum coin{penny, nickel, deme, quarter,  half – dollar, dollr};
Enum coin money;
```

Given this definition and declaration, the following types of statements are valid

```
Money = dime;
If(money ==quarter) printf("is a quarter");
```

The key point to understand about an enumeration is that each of the symbols stand for an integer value and can be used in any integer expression.

For example:

```
printf("the value of the quarter is %d", quarter); is valid.
```

Unless initialized otherwise, the value of the first enumeration symbol is 0, the second is 1 and so on. Therefore,

```
printf("%d%d", penny,dime);
displays 0    2 on the screen.
```

It is possible to specify the value of one or more of the symbols by using an initializer. This is done by following the symbol with an equal sign and an integer value. Whenever an initializer is used, symbols that appear after it are assigned values greater than the previous initialization value For example, the following assigns the value of 100 to quarter.

```
enum coin{penny, nickel, dime, quarter = 100, half – dollar, dollar}:
```

Now the values of these symbols are:

Penny	0
Nickel	1
Dime	2
Quarter	100
Half–dollar	101
Dollar	102

Using initializations, more than one element of an enumeration can have the same value. Symbols of an enumeration cannot be input and output directly. For example, the following code fragment will not perform as desired;

```
/* this will not work */
Money = dollar;
printf("%s", money);
```

remember that dollar is simply a name for an integer: it is not a string. Hence, it is not possible for printf() to display the string "dollar". Likewise, you cannot give an enumeration variable a value using the string equivalent. That is, thiscode does not work:

```
money = "penny";
```

Program:

```
# include<stdio.h>
typedef enum{false, true} boolean;
boolean vowel (char letter)
{
    boolean error = false;
    letter = ('a' < = letter && letter < = 'z')? letter – 32: letter;
    switch(letter)
    {
        case 'A' :
        case 'E' :
        case 'I' :
        case 'O' :
        case 'U" ; break;
        default : error = true;
    }
}

main()
{
    char character;
    do
    {
        printf("Input the character \n");
        scanf("%c", &character);
        getchar();
    }
        while (!vowel (character);
}The tag ie. account is optional in this case.
```

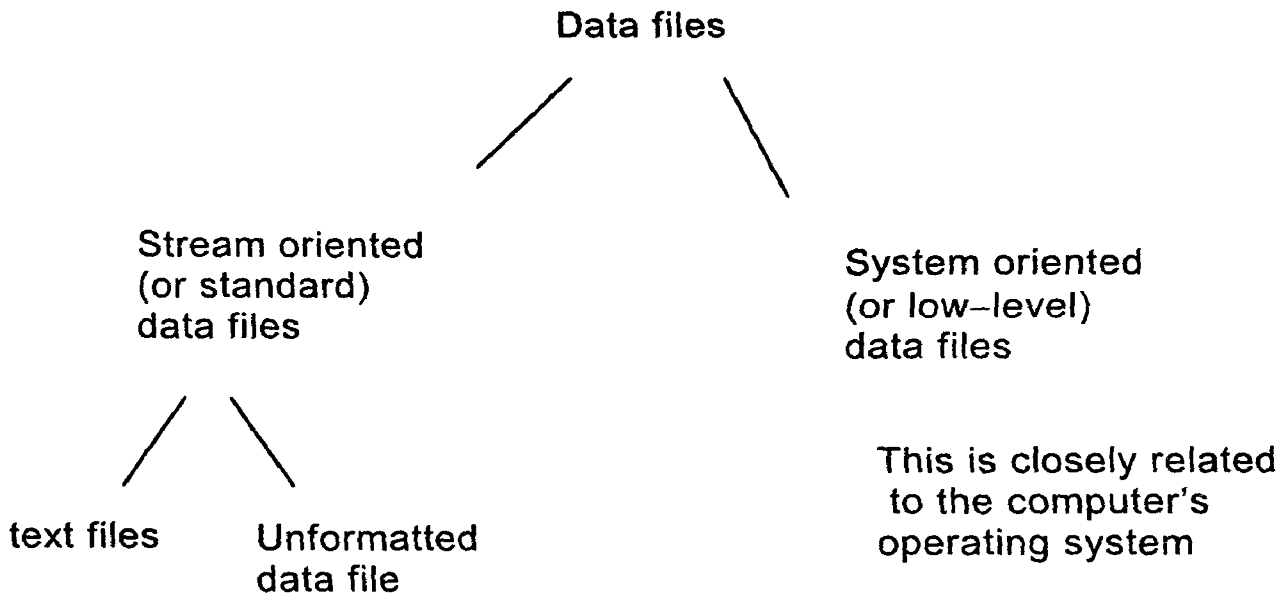
he structure composition will that of the structure variables as shown below:
Id v

UNIT – 8

DATA FILES

Many applications require that information be written to or read from an auxillary memory device. Such information is stored on the memory device in the form of a **data file**.

Thus, data files allow us to store information permanently, and alter that information whenever necessary.



8.1. Opening and closing a Data file:

When working with a stream–oriented data file, the first step is to establish a buffer area, where information is temporarily stored while being transferred between the computer's memory and the data file. This buffer area allows information to be read from or written to the data file.

The buffer area is established by writing

```
FILE    * Ptvar;
```

Where FILE (uppercase letters required) is a special structure type that establishes the buffer area, and ptvar is a pointer variable that indicates the beginning of the buffer area.

A data file must be opened before it can be created or processed. It also specifies how the data file will be utilized, (ie) as a read-only file, a write-only file, or a read / write file, in which both operations are permitted.

The library function f open is used to open a file

This function is typically written as

Ptvar = f open (file–name, file–type);

Where file–name and file–type are strings that represent the name of the data file and the manner in which the data file will be utilized.

File type specification

File type	**Meaning**
"r"	open an existing file for reading only
"w"	open a new file writing only. If a file with the specified file–name currently exists, it will be destroyed and a new file created in its place.
"a"	open an existing file for appending (ie. for adding new information at the end of the file). A new file will be created if the file with the specified file–name does not exist.
"r+"	open an existing file for both reading and writing.
"w+"	open a new file for both reading and writing. If a file with the specified file–name currently exists, it will be destroyed and a new file created in its place.
"a+"	open an existing file for both reading and appending. A new file will be created if the file with the specified file–name does not exist.

The fopen function returns a pointer to the beginning of the buffer area associated with the file. A NULL value is returned if the file cannot be opened as, for example, when an existing data file cannot be found.

Finally, a data file must be closed at the end of the program. This can be accomplished with the library function fclose.

Syntax: fclose (Ptvar);

Program Segment:

```
# include < stdio.h >
FILE  *  fpt;
fpt = fopen ("sample.dat", "w");
_____________________
_____________________

fclose (fpt);
```

Program:

```
# include < stdio.h >
# define NULL 0
```

```
main ()
{
   FILE * fpt;
   fpt = fopen ("Sample.dat", "r+");
   if (fpt == NULL)
     printf ("\n error – cannot open the designated file \n");
else
   {
    ___________
   }
   f close (fpt);
}
```

The fopen and the if statements can also be combined as

```
if ((fpt = fopen ("Sample.dat", "r+")) == NULL)
printf ("\n error – cannot open the designated file \n");
```

8.2 Creating a data file:

A data file must be created before it can be processed. A stream–oriented data file can be created in two ways. One is to create the file directly, using a text editor or a word processor. The other is to write a program that enters information into the computer and then writes it out to the data file.

Program: (creating a data file)

```
/* Lowercase to upper case text conversion */
        # include <stdio.h >
        # include < ctype.h >
        main ()
        {
           FILE * fpt;
           Char c;
            fpt =  fopen ("sample dot", "w");
            /* open a new data file for writing only */
            do
                putc (toupper (c=get char ()), fpt);
                /* read each character and write its uppercase
                equivalent to the data file */
           while (c/= `n');
           fclose (fpt);        /* close the data file */
        }
```

Program: Reading a data file

```
/* read a line of text from a data file and display it on the screen * /
# include <stdio.h >
# include <ctype.h>
# define NULL 0
main ()
{
   File * fpt;
   Char c;
   If (fpt = fopen ("Sample,dat", "r") == NULL)
   /* open the data file for reading only */
   printf (" \n ERROR – cannot open the designated File \n");
   else
        do
               put char (c = get c (fpt));
        while (c ! = `\n');
        fclose (fpt);
}
```

Program: creating a file containing customer records

```
# include <stdio.h>
# include <stdio.h>
# define TRUE 1
typedef struct
   {
        int month;
        int day;
        int year;
   } date;
typed struct
   {
        char Name [80];
        char Street [80];
        char City [80];
        int acc no;
        char acct.type;              /* C (current), 0 (overdue),
                                     or D (delinquent) */
        float old balance;
        float new balance;
        float payment;
        date last payment;
  } record;
```

```
  record read screen (record customer);
  void writefile (record customer);
FILE * fpt;
main ()
{
   int flag = TRUE;  /* variable declaration */
   record customer; /* structure variable declaration */
   fpt = fopen ("records.dot", "w");
   printf ("CUSTOMER BILLING SYSTEM – INITIALIZATION");
   printf ("\n\n enter today 's date (mm/dd/yyyy):");
   Scanf ("%d%d%d",&Customer lastpayment,month,
                        & customer, last payment, day,
                         & customer, last payment, year);
   Customer, newbalance = 0;
    customer  payment = 0;
    customer. acct–type = `C';
    while (flag)
        {
           printf ("\n Name (enter \`END\`when finished):");
           Scanf ("% [^\n]", customer,name);
           printf (fpt,"\n%S\n", Customer name);
           if (strcup (customer,name, "END")==0)
                break;
           customer = read screen (customer);
           writefile (customer);
        }
        fclose (fpt);
}
record readscreen (record customer)
{
    printf ("Street:");
    Scanf ("% [^\n]". customer street);
    printf ("city:");
    Scanf ("%d", & customer.acct–no);
    printf ("current balance:");
    Scanf ("%f", & customer.old balance);
    Return (customer);
}
void writefile (record customer)
{
   fprintf (fpt, "%S\n", customer street);
   fprintf (fpt, "%S\n", customer city);
```

```
   fprintf (fpt, "%d\n", customer, acct.no);
   fprintf (fpt, "%C\n", customer, acct-type);
   fprintf (fpt, "%0.2f\n", customer.old balance);
   fprintf (fpt, "%0.2f\n," customer,new balance);
   fprintf (fpt, "%0.2f\n", customer.payment);
   fprintf (fpt, "%d/%d/%d\n", customer.last payment.month,
                                customer.last payment.day,
                                 customer.last payment.year);
    return;
}
```

Output:

```
CUSTOMER BILLING SYSTEM – INITIALIZATION
enter  today's data (mm/dd/yyyy): 4|21|2004
Name (enter `END' when finished): Janee Wallis
Street: 123 Great white way
City:   New York, NY
Account Number: 701
Current balance: 348.00

Name (enter `END' when finished): John
Street: 1780 Pacific Pork way
City: San Diego, CA
Account number: 658
Current balance: 542.00

Name (enter `END' when finished): George
Street: 45 Alligator Blvd
City: Fort Lauderdale, FL
Account number: 841
Current balance: 257.00
Name (enter `END' when finished): END
```

After the program has been executed, the data file records.dat will have been created containing the following information.

```
Janee Wallis
123 Great white way
New York, NY
701
C
348.00
```

0.00
0.00
4|21|2004
John
1780 Pacific park way
San diego, CA
658
C
542.00
0.00
0.00
4/231/2004
George
45 Alligator Blvd
Fort Lauderdale, FL
841
C
257.00
0.00
0.00
4/21/2004

8.3: Processing a data file: -

Most data file applications require that a data file be altered as it is being processed.

For example, in an application involving the processing of customer records, it may be desirable to add new records to the file, to delete existing records, to modify the contents of existing records, or to rearrange the records.

There are several approaches to this problem. Most obivious approach is to read each record from a data file, updated the record as required, and then write the updated record to the same data file.

Another approach is to work with two different data files, an old file(source) and a new file. Each record is read from the old file, updated as necessary and then written to the new file. When all of the records have been updated, the old file is deleted and the new file renamed. Hence the new file will become the source for the next round of updates.

Updating a file containing customer records:-
Program:-

```
/*update a datafile containing customer records*\
#include<stdio.h>
#include<string.h>
#define NULL 0
#define  TRUE 1
typedef struct
{
   int month;
   int day;
   int year;
} date;
typede struct
{
   char name[80];
   char street[80];
   char city[80];
   int acct_no;
   char acct_type;    /*(current, o(overdue),D(delinguent)*\
   float oldbalance;
   float newbalance;
   float payment;
   date last payment;
} record;
record read file (record customer);
record update (record customer);
record write file (record customer);
FILE *ptold, *ptnew;
int month, day year;
main( )
{
   int flag = TRUE;
   record customer;
   if((ptold=fopen("records.old","r"))==NULL)
   printf("\nERROR-cannot open the designated read file\n");
   else
    {
    ptnew=fopen ("records.new","w");
    printf ("CUSTOMER BILLING SYSTEM – update");
    printf("\n\n enter today \s' date (MM\dd\yyyy):");
    scanf("%d/%d/%d", &month, &day, &year);
```

```
        while(flag)
        {
            fscanf(ptold,"%[^\n]", customer.name);
            fprintf(ptnew, "\n%s\n", customer.name);
            if(stromp (customer.name, "END")==0)
                          break;
            customer = readfile(customer);
            customer = update (customer);
            writefile(customer);
        }
      fclose(ptold);
      fclsose(ptnew);
    }
}
record readfile (record customer)
{
    fscanf(ptold,"%[^\n]",customer.struct);
    fscanf(ptold,"%[^\n]",customer.city);
    fscanf(ptold,"%d",&customer.acct-no);
    fscanf(ptold,"%c",&customer.acct-type);
    fscanf(ptold,"%f",&customer.oldbalance);
    fscanf(ptold,"%f",&customer.newbalance);
    fscanf(ptold,"%d",&customer.payment);
                    &customer.lastpayment.month,
                    &customer.lastpayment.day,
                    &customer.lastpayment.year,
    return(customer);
}
record update(record customer)
{
    printf("\n\n Name: %s", customer.name);
    printf("Account number: %d\n", customer.acct_no);
    printf("\n oldbalance= %7.2f", customer.old balance);
    printf("Current payment:");
    scanf("%f",&customer.payment);
    if(customer.payment>0)
    {
        customer.last payment. Month = month;
        customer.last payment. day = day;
        customer.last payment. Year = year;
        customer.Acct_type=(customer.payment<0.1*
                                custmer.oldbalance)?'O':'C';
```

```
    }
    else
        customer.acct_type:(customer.oldbalance>o)? 'D': 'C';
        customer newbalance=customer.oldbalance _customer.payment;
        printf("New balance :%7.2f",customer.newbalance);
       printf("Account status:");
       switch(customer.acct_type)
       {
           case 'C':
                   printf("CURRENT\n");
                   break;
           case'O':
                   printf("OVERDUE\n");
                   break
           case'D':
                   printf("DELINQUENT\n");
                   break;
           default:
                   printf("ERROR\n");
      }
           return(customer);
   }
   void write file(record customers)
   {
          fprint(ptnew."%s\n",customer, street);
          fprint(ptnew."%s\n",customer, city);
          fprint(ptnew."%d\n",customer, acct_no);
          fprint(ptnew."%c\n",customer, acct_type);
          fprint(ptnew."%0.2f\n",customer, oldbalance);
          fprint(ptnew."%0.2f\n",customer, newbalance);
          fprint(ptnew."%0.2f\n",customer, payment);
          fprint(ptnew."%d\%d\%d\n",customer.lastpayment.month,customer.
                              lastpayment.day,customer.lastpayment.year);
          return;
          }
```

Each customer name is read from the source file and then written to the new file within main. The remaining information for each record is then read from the source file, updated and written to the new file within the functions read file, update and write file respectively. This process continues until a record is encountered containing the customer name END. Both data files are then closed, and the computation terminates.

CUSTOMER BILLING SYSTEM – UPDATE
Enter today's date (mm/dd/yyyy): 10/22/2004
Name: Janee wallis Account number:701
Oldbalance: 348.00 current payment:30.00
New balance:318.00 Account status: OVERDUE
Name: John Account number: 658
Old balance: 542.00 current payment:55
New balance: 487.00 Account status:CURRENT
Name: George Account number:841
Old balance: 257.00 current payment: 0
New balance 257.00 Account status: DELINQUENT

After all of the customer records have been processed the new data file records new will have been created, continuing the following information.

Jance wallis
123 Great while way
Newyork; NY
701
0
348.00
318.00
30.00
10/22/2004
John
1780 pacific part way
san Diego, CA
658
C
542.00
487.00
55.00
10/2/22004
George
45 Alligator Blvd
Fort Landerdale,FL
841
D
257.00
257.00
0.00
10/22/2004

8.4: Unformatted data files:-

A data file may consist of multiple structures having the same composition, or it may contain multiple arrays of the same type and size. For such applications it may be desirable to read the entire block from the data file, or write the entire block to the data file, rather than reading or writing the individual components (ie. structure members, or array elements) with in each block separately.

The functions fread and fwrite are the unformatted read and write functions.

A typically fwrite function is written as

```
fwrite(&Customer, size of (record) ,1, fpt);
```

Where customer is a structure variable of type record and fpt is the stream pointer associated with a data file that has been opened for output.

Creating an unformatted data file containing customer records:-

```
#include<stdio.h>
#include<string.h>
#define TRUE 1
typedef struct
{
     int month;
     int day;
     int year;
} date;
  typefstruct
  {
       char name[80];
       char street[80];
       char city[80];
       int accty_no;
       char acct_type;
       float oldbalance;
       float newbalance;
       float payment;
       date lastpayment;
    }record;
record read screen(record customer);
FILE *fpt;
Main()
{
```

```
int flag =TRUE;
record customer;
fpt = fopen("data.bin" ,"w");
printf("customer Billing System – Initialzation\n");
printf("enter today's date (mm\dd\yyyy)::");
scanf("%d/%d/%d",&customer.lastpayment.month,
      &customer.lastpayment.day), &customer.lastpayment .year);
customer.new balance = o;
customer.payment = o;
customer.acct_type='c';
while (flag)
{
   printf("\nName(enter\'END\' when finished):");
   scanf("%[^\n]",customer.name);
   if(strump(customer.name,"END")==o)
      break;
   customer = read screen(customer);
   fwrite(&customer,size(Record),1,fpt);
   /*erase strings*/
   strset(customer.name, ' ' );
   strset(customer.street, ' ' );
   strset(customer.city, ' ' );
}
   fclose(fpt);
}
 record read screen(record customer)
 {
   printf("street:");
   scanf("%[ ^\n];customer, street);
   printf("city:");
   scanf("%[ ^\n];customer, city);
   printf("Account number:");
   scanf("%d", &customer, acct_no);
   printf("current balance:");
   scanf("%f", &customer.oldbalance);
   return(customer);
 }
```

Comparing this program with that shown in **creating a file containing customer records,** we see that the two program are very similar. Within main, the present program reads each customer name and tests for a stopping condition(END), but does not write the customer name to the data file, as in the earlier program. Rather, if a stopping condition is not indicated, the present program reads the

remainder of the customer record to the data file with the, since the fwrite library function takes its place.

After each record has been written to the data file, the string members customer.name, customer. street, customer.city are cleared. The library functions strset is used for this purpose.

Strset(customer.name, ' '); courses the contents of customer.name to be replaced with blank characters. Note that the header file strin.h is included in this program, in suppo. cf the strset function.

Once an unformatted data file has been created, how to detect an end_of_file condition. The library function feof is available for this purpose. Actually, feof will indicate an end_of_file condition for any stream_oriented data file, not just an unformatted data file. This function returns a non-zero value (TRUE) if an end_of_file condition has been detected, and a value of zero(FALSE) if an end_of_file is not detected. Hencea program that reads an unformatted data file can utilize a loop that continues to read successive records, as long as the value returned by feof is not TRUE.

Program:-Updating an unformatted data file containing customer records:-

```
#include<stdio.h>
#define NULL 0
typedef struct
{
    int month;
    int day;
    int year;
} date;
  typefstruct
  {
      char name[80];
      char street[80];
      char city[80];
      int accty_no;
      char acct_type;
      float oldbalance;
      float newbalance;
      float payment;
      date lastpayment;
    }record;
record update(record customer);
```

```
FILE *ptold, *ptnew;
int month,day,year;
Main()
{
        record customer;
        if((ptold = fopen("data.old","r")) ==NULL)
        printf("\nERROR – cannot open the designated readfile\n");
        else
        {
        ptnew = fopen("data,new","w");
        printf("CUSTOMER BILLING SYSTEM – UPDATE\n\n");
        printf("enter today\'sdate(mm/dd/yyy);
        scanf("%d/%d/%d",&month,&day, &year);
        fread(&customer,size of(record), 1,ptold);
        while(ifeof(ptold))
        {
           customer = update(customer);
           fwrite(&customer,size of(record), 1,ptnew);
           fread(&customer,size of(record), 1,ptold);
           } /*end while*\
            fclose(ptold);
            fclose(ptnew);
             }  /*end else*\
           }  /*end main*\
record update(record customer)
    {
        printf("\n\n Name: %s", customer.name);
        printf("Account number: %d\n", customer.acct_no);
        printf("\n oldbalance= %7.2f", customer.old balance);
        printf("Current payment:");
        scanf("%f",&customer.payment);
        if(customer.payment>0)
    {
        customer.lastpayment.month = month;
        customer.lastpayment.day = day;
        customer.lastpayment.year = year;
        customer.acct_type=  (customer.payment<
                                     0.1*customer.oldbalance)? 'o' : 'c';
    }
        else
```

```
    customer.acct_type = (customer.oldbalance>0) ?'D' :'c';
        customer.newbalance=customer.oldbalance –
                                        customer.payment;
    printf("Newbalance: %7.2f", customer.new balance);
    printf("Account status:")
    switch(customer.acct_type)
    {
    case 'C':
            printf("CURRENT\n");
            break;
    case'O':
            printf("OVERDUE\n");
            break
    case'D':
            printf("DELINQUENT\n");
            break;
    default:
            printf("ERROR\n");
  }
    return(customer);
}
```

UNIT – 9

QUEUES

9.1: The Queue and its sequential representation

A queue is an ordered collection of items from which items may be deleted at one end (called the front of the Queue) and into which items may be inserted at the other end (called the rear of the queue).

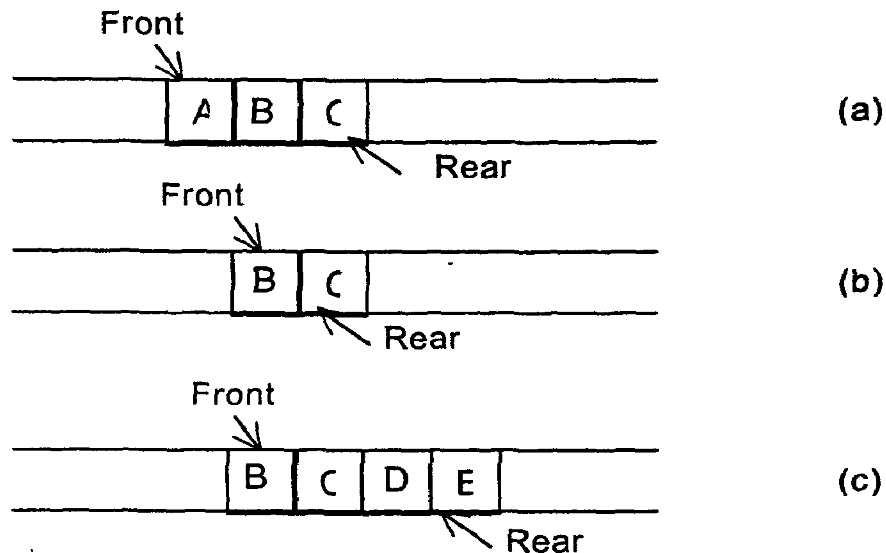


Figure (a) illustrates a queue containing 3 elements A B,C.

A is at the front of the queue and c at the rear. In figure (b) an element has been deleted from the queue. Since elements may be deleted only from the front of the queue, A is removed.

In figure (c), when items D and E are inserted, they must be inserted at the rear of the queue.

Since D was inserted into the queue before E, it will be removed earlier. The first element inserted into a queue is the first element to be removed. For this reason a queue is sometimes called a **fifo** (first in first out) list as opposed to a stact which is **lifo** (last in, first out).

A line at a bank, line at a bus stop and group of cars waiting at toll booth are all familiar examples of queue.

Three primitive operations can be applied to a queue.

insert (q,x)

insert item x at the rear of the queue q.

x = remove (q)

deletes the front element from the queue q and sets x to its contents.

empty (q)

returns false or true depending on whether or not the queue contains any elements.

The queue in the above figure can be obtained by the following sequence of operations.

We assume that the queue is initially empty.

```
insert (q,A);
insert (q,B);
insert(q,C);            figure (a)
x= remove (q);          figure (b); x is set to A
insert (q,D);
insert (q,E);           figure (c);
```

The **insert** operation can always be performed. Since there is no limit to the number of elements a queue may contain.

The **remove** operation, however, can be applied only if the queue is no empty. The result of an illegal attempt to remove an element from an empty queue is called **underflow**. The empty operation is, of course, always applicable.

9.2 Queue as an abstract data type:

We use **eltype** to denote the type of the queue and parameterize the queue type with **eltype**.

```
abstract typedef <<eltype>> QUEUE (eltype);
abstract empty (q)
QUEUE (eltype) q;
post condition empty == (len (q) ==0);

abstract eltype remove (q)
QUEUE (eltype)q;
precondition empty (q) = = FALSE;
```

post condition remove = = first (q');
q = = sub (q', 1, len(q')– 1);

abstract insert (q, elt)
QUEUE (eltype) q;
eltype elt;
postcondition q== q' + <elt>;

9.3: C implementation of Queues:

To represent a queue in C, we can use an array to hold the elements of the queue and to use two variables, **front** and **rear**, to hold the position within the array of the first and last elements of the queue.

We might declare a queue q of integers by

```
#define MAXQUEUE 100
struct queue{
        int items [MAXQUEUE];
        int front, rear;
}q;
```

Of course, using an array to hold a queue introduces the possibility of **overflow** if the queue should grow larger than the size of the array. Ignoring the possibility of underflow and overflow for the moment, the operation **insert (q,x)** could be implemented by the statements

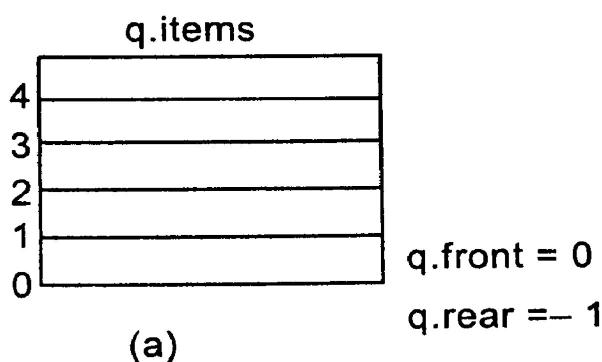
```
q.items [++ q.rear] = x;
```

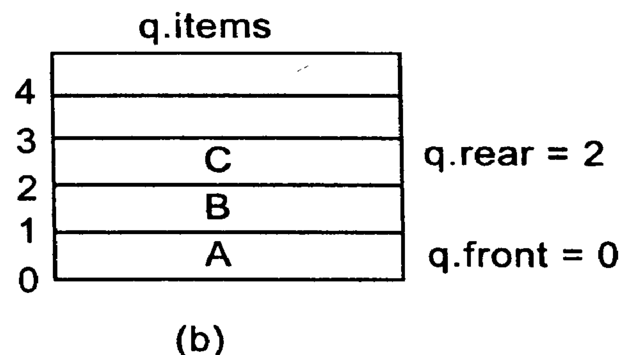
and the operation x = remove(q) could be implemented by

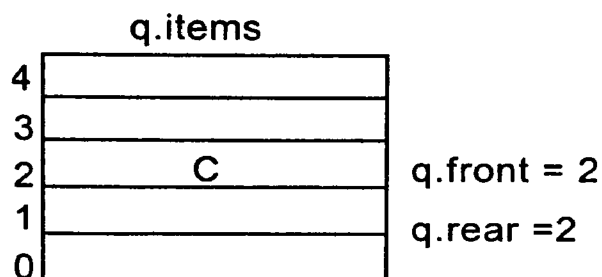
```
x=q. items[q.front ++];
```

Initially, q.rear is set to –1, and q.front is set to 0. The queue is empty whenever q.rear <q.front. The number of elements in the queen at any time is equal to the value of

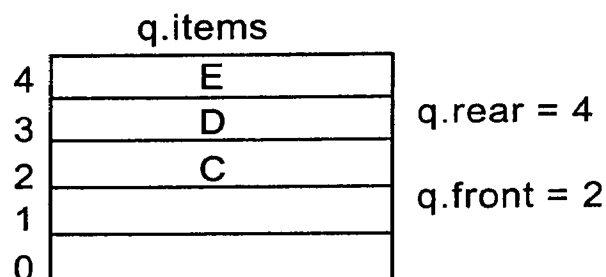
q.rear – q.front + 1







(c)



(d)

Here MAXQUEUE equals 5. Intially (figure(a)) the queue is empty. In figure (b) items A,B,C have been inserted. In figure (c) two items A,B have been deleted. In figure (d) two new items D,E have been inserted.

Here the value of q.front is 2 and q.rear is 4, so that there are 4 – 2 +1=3 elements in the queue.

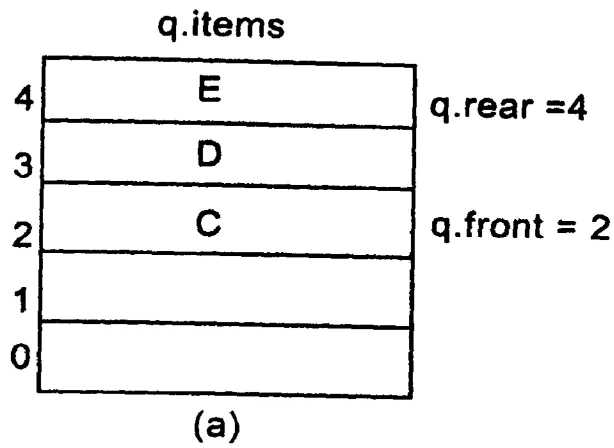
However to insert F into the queue, q.rear must be increased by 1 to 5 and q.items[5] must be set to the value F. But q.items is an array of only five elements, so that the insertion cannot be made.

To overcome this we can modify the **remove** operation so that when an item is deleted, the entire queue is shifted to the beginning of the array. The operation x=remove(q) would then be modified (again, ignoring the possibility of underflow) to

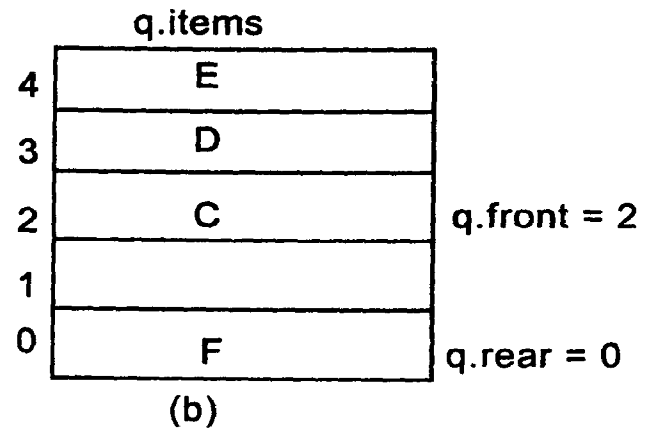
```
x = q.items[0];
for (i=0; i<q.rear; i++)
        q.items[i] = q.items[i+1];
q.rear – –;
```

This method, however, is too inefficient. Each deletion involves moving every remaining element of the queue. If a queue contains 500 or 1000 elements, this is clearly too high a price to pay.

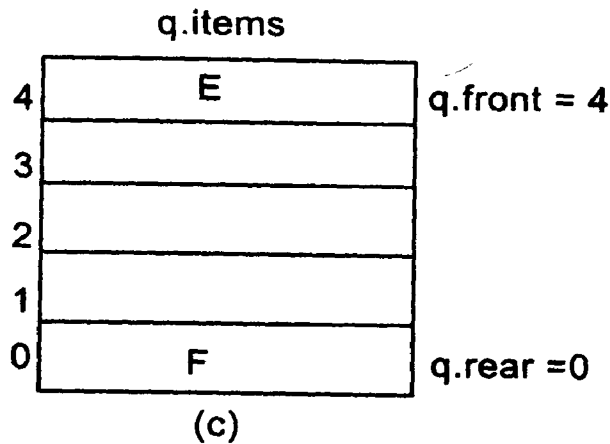
Another way is to view the array that holds the queue as a circle rather than as a straight line. That is, we imagine the first element of the array (ie, the element at position 0) as immediately following its last element. This implies that even if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty.



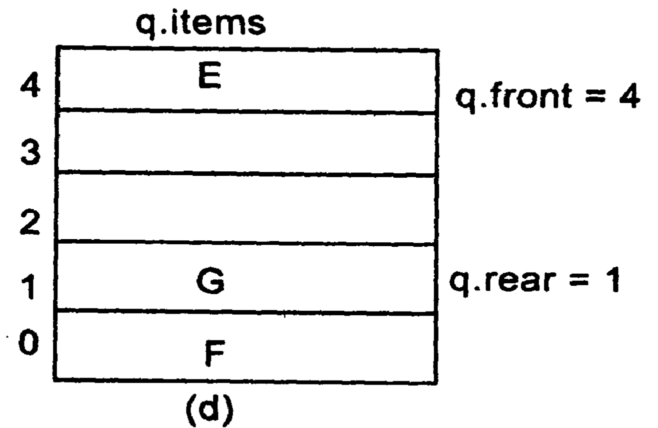
(a)



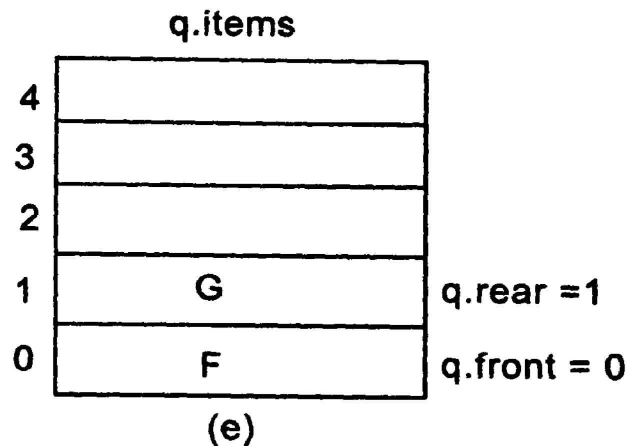
(b)



(c)



(d)



(e)

In figure (a) a queue contains 3 elements in positions 2,3,4 of a 5 element array.

Although the array is not full, its last element is occupied. If item F is now inserted into queue, it can be placed in position 0 of the array as shown in figure (b).

In figure (c) the first two items C and D are delected.

In figure (d) G is inserted and finally in figure (e) E is deleted.

Unfortunately, it is difficult under this representation to determine when the queue is empty. The condition **q.rear < q.front** is no longer valid as a test for the empty queue.

One way of solving this problem is to establish the convention that the value of q.front is the array index immediately preceding the first element of the queue rather than the index of the first element itself. Thus since q.rear is the index of the last element of the queue, the condition

q.front == q.rear implies that the queue is empty. A queue of integers may therefore be declared and initialized by

```
#define MAXQUEUE 100
struct queue
{
   int items [MAXQUEUE];
   int front, rear;
}
struct queue q;
q.front = q.rear = MAXQUEUE–1;
```

Note that q.front and q.rear are initialized to the last index of the array, rather than to –1 or 0, because the last element of the array immediately precedes the first one within the queue under this representation.

Since q.rear equals q.front, the queue is initially empty

The empty function may be coded as

```
int empty (struct queue *pq)
{
    return ((pq → front ==pq → rear) ? TRUE : FALSE);
} /* end empty */
```

Once this function exists, a test for the empty queue is implemented by the statement

```
if (empty ( &q))
   /* queue is empty */
else
   /* queue is not empty */
```

The operation remove (q) may be coded as

```
int remove (struct queue *pq)
{
     if(empty (pq))
     {
        printf("queue underflow");
        exit(1);
     }
     if (pq → front == MAXQUEUE – 1)
        pq → front = 0;
     else
        (pq → front) ++;
        return (pq → items [pq → front]);
} /* end remove */
```

Note that pq is already a pointer to a structure of type queue, so the address operator "&" is not used in calling empty within remove. Also note that pq→ front must be updated before an element is extracted.

An underflow condition is meaningful and serves as a signal for a new phase of processing.

The function remvandtest is

```
Void remvandtest (struct queue *Pq, int *px, int *pund)
```

If the queue is not – empty, this routine sets *pund to FALSE and * px to the element removed from the queue.

If the queue is empty, so that underflow occurs, the routine sets *pund TRUE.

9.4: Priority Queue:

The **priority queue** is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations

There are two types of priority queues

1. An ascending priority queue
2. A descending priority queue

An **ascending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.

If apq is an ascending priority queue, the operation **pq insert (apq, x)** inserts element x into apq and **pqmindelete (apq)** removes the minimum element from apq and return its value.

A **descending priority queue** is similar but allows deletion of only the largest item. The operations applicable to a descending priority queue are **dpq, pqinsert(dpq, x)** and **pqmaxdelete(dpq)**.

pqmax delete (dpq) removes the maximum element from dpq and returns its value.

The operation **empty(pq)** applies to both types of priority queue and determines whether a priority queue is empty.

Once pqmindelete has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest, and so on.

Similarly, pqmaxdelete retrieves elements of a descending priority queue in descending order.

Once pqmindelete has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest, and so on.

Similarly, pqmaxdelete retrieves elements of a descending priority queue in descending order.

The elements of the priority queue need not be numbers or characters that can be compared directly. They may be complex structures.

As we have seen, a stack and a queue can be implemented in an array so that each deletion or insertion involves accessing only a single element of the array. Unfortunately, this is not possible for a priority queue.

Suppose we attempt the operations pqnumdelete(pq) on an ascending priority queue. This raise two issues:

First, to locate the smallest element, every element of the array from pq.item[0] through pq.items [pq.rear –1] must be examined. Therefore a deletion requires accessing every element of the priority queue.

Second, how can an element in the middle of the array be deleted. Stack and queue deletions involve removal of an item from one of two ends and do not require any searching. Priority queue deletion under this implementation requires both searching for the element to be deleted and removal of an element in the middle of an array.

There are several solutions to this problem, none of them entirely satisfactory:

1. A special "empty" indicator can be placed into a deleted position. This indicator can be a value that is invalid as an element, or a separate field can be contained in each array element to indicate whether it is empty, Insertion proceeds as before, but when pq.rear reaches maxpq the array elements are compacted into the front of the array and pq.rear is reset to one more than the number of elements. There are several disadvantages to thiis approach.

   (a) The search process to locate the maximum or minimum element must examine all the deleted array positions in addition to the actual priority queue elements. If many items have been deleted but no compaction has jet taken place, the delection operation accesses many more array elements than exist in the priority queue.
   (b) Once in a while insertion requires accessing every single position of the array as it runs out of room and begins compaction.

2. The deletion operation labels a position empty as in the previous solution, but insertion is modified to insert a new item in the first "empty" position. Insertion then involves accessing every array element up to the first one that has been deleted. This decreased efficiency of insertion is a major drawback to this solution.

3. Each deletion can compact the array by shifting all elements past the deleted element by one position and then decrementing pq.rear by 1. Insertion remains unchanged. On the average, half of all priority queue elements are shifted for each deletion, so that deletion becomes quite in efficient. A slightly better alternative is to shift either all preceding elements forward or all succeding elements backward, depending upon which group is smaller.

4. Instead of maintaining the priority queue as an unordered array, maintain it as an ordered, circular array.

UNIT – 10
LINKED LIST

Draw back of using sequential storage to represent stacks and queues is that a fixed amount of storage remains allocated even when the structure is actually using a smaller amount or possibly no storage at all. Further, no more than that fixed amount of a storage may be allocated, thus introducing the possibility of overflow.

Assume that a program uses two stacks implemented in two separate arrays.

S1.items & S2.items

Further assume that each of these arrays has 100 elements. Then despite the fact that 200 elements are available for the two stacks, neither can go beyond 100 items. Even if the first stack contains only 25 items, the second cannot contain more than 100.

One solution to this problem is to allocate a single array items of 200 elements. The first stack occupies items (0), items(1),... items (top1), while the second stack is allocated from the other end of the array, occupying items(199), items (198).... Items(top2). Thus when one stack is not occupying storage the other stack can use that storage of course, two distinct sets of pop, push and empty routines are necessary for the two stacks, since one grows by incrementing top1, while the other grows by decrementing top2.

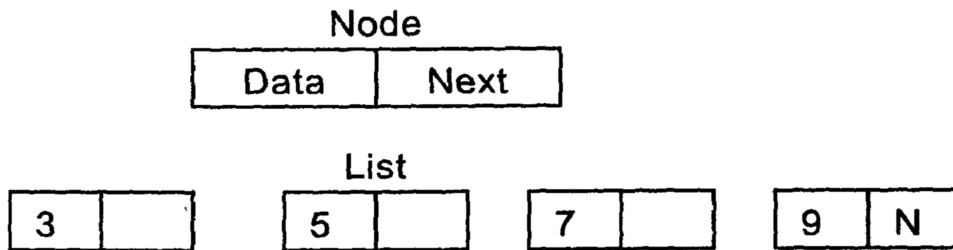
In a sequential representation, the items of a stack or queue are implicitly ordered by the sequential order of storage.

10.1:Linked Lists:-

List is a homogeneous collection of elements with linear relationship between the elements. In linked list representation, data elements are not constrained to be stored in sequential order, rather the individual elements are stored "somewhere" in the memory. The order of elements is maintained by explicit links between them. Thus 1 each node in a linked list contains atleast two fields:

- Information fields: which contains the user's data
- Next fields: Contains pointer to the next node in the list.

Pictorial representation of linked list



Advantage of Linked List:-

A linked list is a dynamic data structure. Therefore, the primary advantage of linked lists over array is that lists can grow or shrink in size during the executing of a program. The second advantage of linked list is that it does not waste memory space. The third advantage of linked list is that the linked lists provide flexibility in allowing the items to be rearranged efficiently

Disadvantage of Linked list

The major limitation of linked list is that the access to any arbitrary item is time consuming.

10.2: Operation on Linked list:-

The Linked list as an abstract data type and perform the following basic operations.

- Inserting an item
- Traversing the list
- Counting the item in the list
- Printing the list
- Looking up an item for editing or printing
- Deleting an item.

Inserting an item:-

Inserting a new item say x into the list has three situations.

1. Insertion at front of the list
2. Insertion in the middle of the list
3. Insertion at the end of the list.

The process of insertion precedes a search for the place of insertion. The search involves in locating a node a after which the new item is to be inserted. A general algorithm for insertion is as follows:

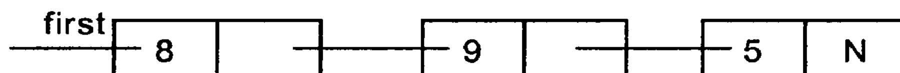
```
Begin
    If the list is empty or the new node comes before the head node then
            Insert the new mode as head node
    Else
            If the new node comes after the last node then
                    Insert the new node as the end node
    Else
                    Insert the new node in the body of the list.
    End
```

Algorithm for placing the new item at beginning of a Linked list

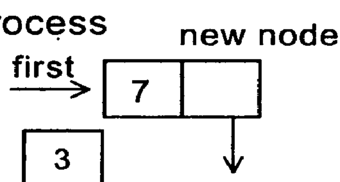
1. Set space for new node x
2. Assign data to information field of the new node.
3. Set the next field of the new node to point to the start (head node) of the list.
4. Change the head pointer to point to the new node.

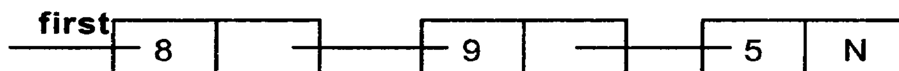
Insert operation (1) – list

Before Insertion



Insertion process





After Insertion

Program for the above algorithm

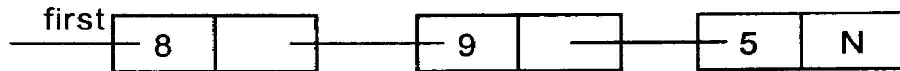
```
Insert Beg(int value)
{
        node *new node = (node*)malloc (size of (struct node));
        new node → data  =  value;
        new node → next  =  first; /*first is the pointer to list*/
        first = new node;
}
```

Algorithm for inserting the new node x between two existing nodes N1 and N2

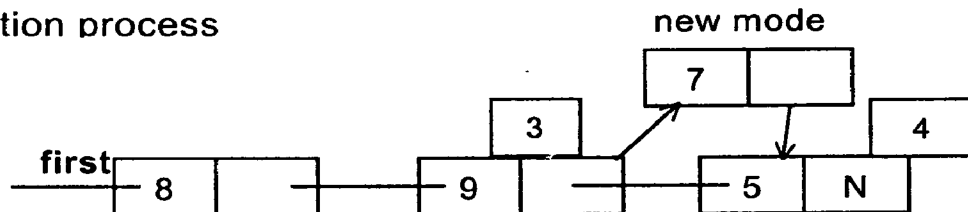
1. Set space for new node x
2. assign value to information field of x
3. set the next field to point to node N2
4. set the next field of NI to point to X.

Insert operation (2) – list

Before Insertion



Insertion process



After Insertion

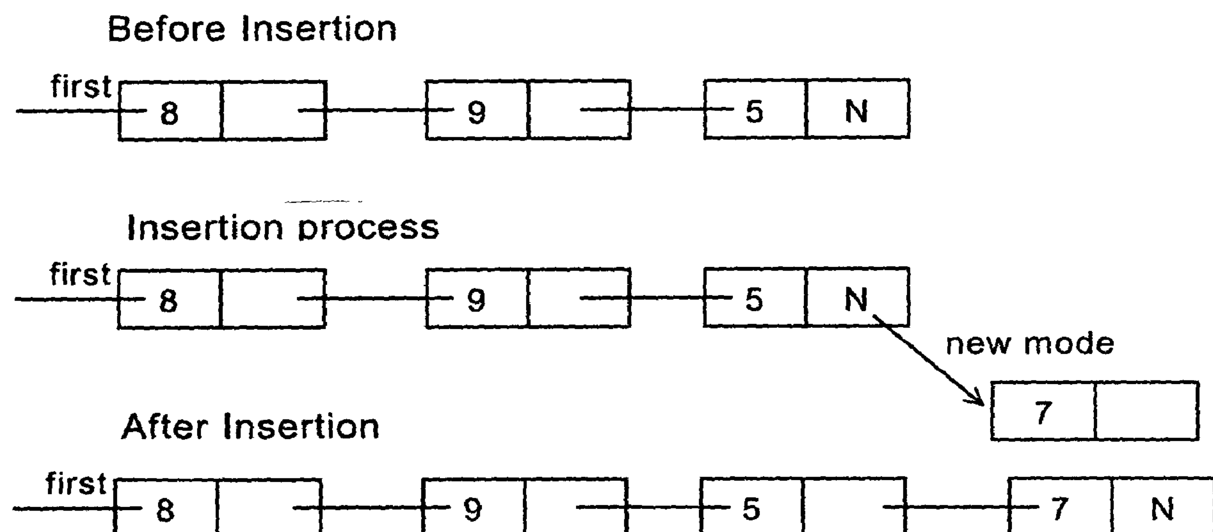
Program for the above algorithm: -

```
InsertBet (int value)
{
        node *temp.*prev;
        node*new node = (node*) malloc (size of (struct node));
        new node → data = value;
        new node → next = NULL;
                /* search the position for insertion*/
        temp = first;
        prev = NULL;
        while(temp! = NULL)
          {
                if (value <temp → data)
                  {
                        prev = temp
                        temp = temp → next;
                else
                        break;
```

```
            } /* end of search process*/
            /*insert process*/
            if(prev=NULL) /*insert at the beginning*/
            {
                newnode → next = first;
                first = newnode;
            }
        else
         {      /*insert at the middle of the list*/
            newnode → next = prev → next
            prev → next node
         }
    }
```

Algorithm for inserting an item at the end of the list is similar to the one for inserting in the middle, except the next field of the now node is set to null.

Append operation List



Program for the above algorithm:-

```
    Inserted (int value)
      {
          node *temp;
          node *newnode = (node *) malloc(size of (Struct node;))
          new node → data = value;
          new node → next = NULL;
          /* segment to move to the last node in the list*/
          temp = first;
```

```
        while (temp →next!=NULL)
    {
        temp = temp →next;
    }
        temp → next = new node;
}
```

Deletion operation:-

Deleting node from a list is easier than insertion process, as only one pointer value need to be changed. Here again we have 3 situation

1. Deleting the first node
2. Deleting the last node
3. Deleting a node between two nodes in the middle of the list.

In the first case, the head pointer is altered to point to the second node in the list in other two cases, the pointer pf the node immediately preceding the one to be deleted is altered to point to the node following the deleted node.

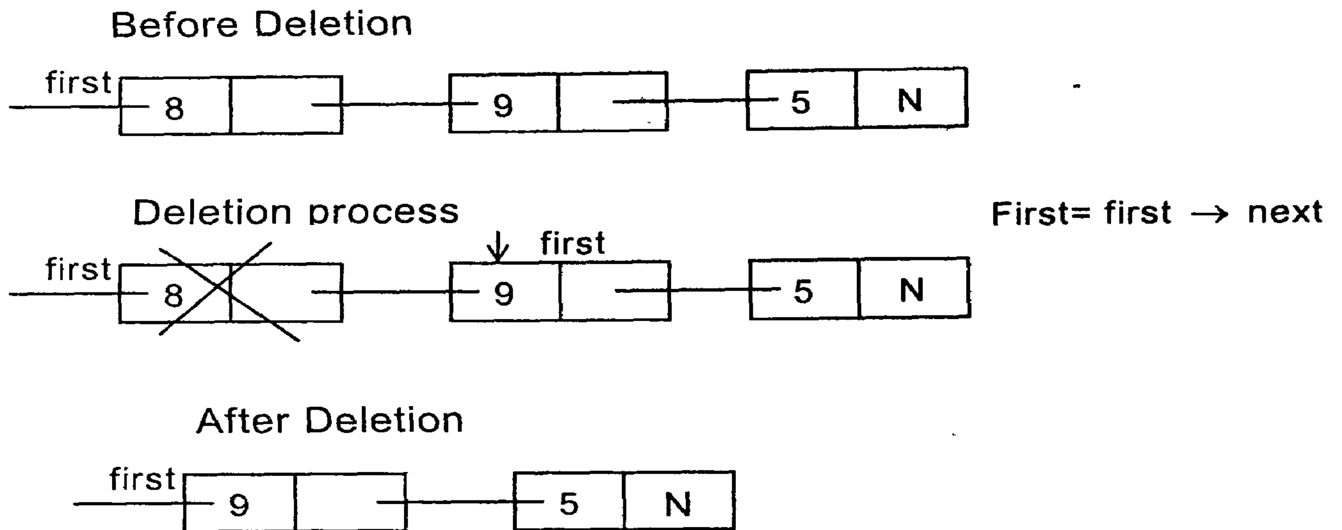
The general algorithm for deleting is as follows:-

```
Begin
        If list is empty then
                Node cannot be deleted
Else
        If node to be deleted is the first
Order then
        Make the head to point to the second node.
Else
        Delete the node from the body of the list
End
```

Program to delete the first node in the list

```
Deletefirst (intvalue)
{
  first = first→next;
}
```

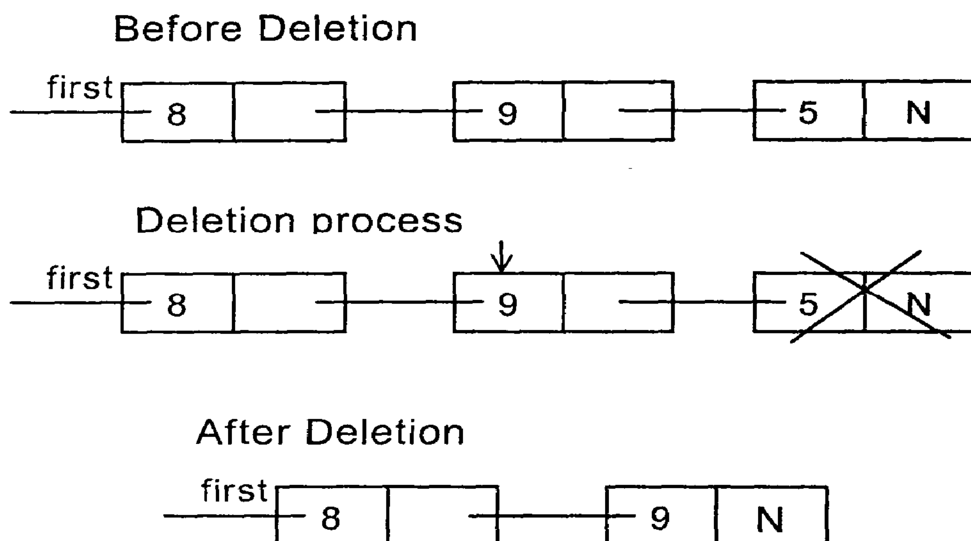
Delete operation (1) – List



Program to delete the last node in the list.

```
Deletefirst(int value)
{
        node*temp.*prev;
        /*segment to more to the last node in the list*/
        prev = NULL;
        temp = first;
        while(temp→next! = NULL)
           temp = temp → next;
        prev → next = NULL; /*deleting a node*/
}
```

Delete operation (2) – List



10.3: List implementation of priority queues:-

An ordered list can be used to represent a priority queue.

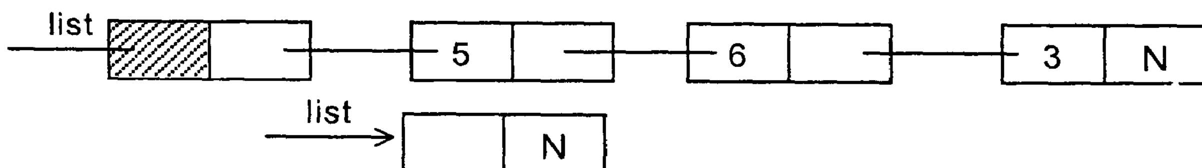
For an ascending priority queue, insertion is implemented by the **place** operation, which keeps the list ordered, and deletion of the minimum element is implemented by the pop operation which removes the first element from the list. A descending priority queue can be implemented by keeping the list in descending, rather than ascending order or by using remove to implement pqmax delete

An unordered list may also be used as a priority queue. Such a list requires examining only one node for insertion but always required examining n elements for deletion.

Thus an ordered list is somewhat more efficient than an unordered list in implementing a priority queue.

10.4: Header nodes:-

Sometimes it is desirable to keep an extra node at the front of a list such a node does not represent an item in the list and is called a header node. The information field of such a header node niht be unused. More often, the information field (into portion) of such a node could be used to keep global information about the entire list.



Another possibility for the use of the info portion of a list header is as a pointer to a current node in the list during a traversal process.

10.5: Lists in C : Array implementation of lists:-

How can linear lists be represented in C? since a list is simply a collection of nodes, an array nodes immediately suggests itself. However, the nodes cannot be ordered by the array ordering , each must contain within itself a pointer to its successor.

In this scheme a pointer to a node is represented by an array index. (ie)A pointer is an integer between 0 and NUMNODES – 1 that references a particular element of the array node. The null pointer is represented by the integer –1.

		Info	next
	0	26	–1
	1	11	9
	2	5	15
list 4 =	3	1	24
list 2 =	4	17	0
	5	13	1
	6		
	7	19	18
	8	14	12
	9	4	21
	10		
list 3 =	11	31	7
	12	6	2
	13		
	14		
	15	37	23
list 1 =	16	3	20
	17		
	18	32	–1
	19		
	20	7	8
	21	15	–1
	22		
	23	12	–1
	24	18	5
	25		
	26		

Array.of nodes containing four linked list

Above is a portion of an array node that contaıns four linked lists.

The list list 1 start at node [16] and contains the integers 3,7,14,6,5,37,12. The nodes that contain these integers in their info field are scattered through out the array. The next field of each node contains the index within the array of the node containing the next element of the list. The last node on the list is node [23], which contains the integer 12 in its info field and the null pointer (–1) is in its next field, to indicate that it is last on the list.

Similarly, list 2 begins at node [4] and contains the integers 17 and 26, list 3 begins at node[11] and contains the integers 31,19,32 and list 4 begins at node [3] and contains the integers 1,18,13,11,4 and 15.

The variables list1, list2, list3 and list4 are integers representing external pointers to the four lists.

10.6: simulation using linked lists:-

One of the most useful applications of queues, Priority queues and linked lists is in simulation.

Simulation is the use of one system to imitate the behaviour of another system. Physical simulations such as tunnels used to experiment with designs for carbodies and flight simulation used to describe airline pilot.

Suppose that there is a bank with four tellers. A customer enters the bank at a specific time (t1) desiring to conduct a transaction with any teller. The transaction may be expected to take a certain period of time (t2) before it is completed. If a teller is free, the teller can process the customer's transaction immediately, and the customer leaves the bank as soon as the transaction is completed, at time t1+t2. The total time spent in the bank by the customer is exactly equal to the duration of the transaction (t2).

However it is possible that none of the tellers are free; they are all servicing customers who arrived previously. In that case there is a line waiting at each teller's window the line for a particular teller may consist of a single person the one currently transacting business with the teller – or it may be a very long line. The customer proceeds to the back of the shortest line and waits until at1 the previous customers have completed their transactions and have left the bank. At that time the customer may transact his or her business. The customer leaves the bank at t2 time after reaching the front of a teller's line. In this case the time spent in the bank is t2 + the time spent waiting on line.

Give such a system, we would like to complete the average time spent by a customer in the bank. One way of doing so is to stand in the bank doorway, ask departing customers the time of their arrival and record the time of their departure, subtract the first from the second for each customer, and take the average over all customers. However, this would not be very practical. It would be difficult to ensure that no customer is overlooked leaving the bank.

Instead, we write a program to stimulate the customer actions. Each part of the real world stimulation has its analogue in the program. The real world

action of a customer arriving in modeled by input of data. As each customer arrives, two facts are known the time of arrival and the duration of transaction. Thus the input data for each customer customer consists of a pair of numbers: the time of the customer's arrival and the amount of time necessary for the transaction. The data pairs are ordered by increasing arrival time. We assume atleast one input line.

The four lines in the bank are represented by four queues. Each node of the queues represents a customer waiting on a line, and the node at the front of a queue represents the customer currently being serviced by teller.

Simulation process:-

The simulation proceeds by finding the next even to occur and effecting the change in the queues that mirrors the change in the lines at the bank due to that event. To keep track of events, the program uses an ascending priority queue, called the event list. This list contains at most five nodes, each representing the next occurrence of one of the five types of events. Thus the event list contains one node representing the next customer arriving and four nodes representing each of the 4 customers at the head of a line completing a transaction and leaving the bank. Of course, it is possible that one or more of the lines in the bank are empty, or that the doors of the bank have been closed for the day, so that no more customers are arriving. In such cases the event list contains fewer than 5 nodes.

An event node representing a customer's arrival in called an arrival node, and a node representing a departure is called a departure node. At each point in the simulation, it is necessary to know the next event to occur. Thus the even list is an ascending priority queue represented by an ordered linked list.

The first event to occur is the arrival of the first customer. The event list is therefore intialised by reading the first input line and placing an arrival node representing the first customer's arrival on the event list. Initially, of course all four teller queues are empty. The simulation proceeds as follows; the first node on the event list is removed and the changes that the event causes are made to the queues. As we shall soon see, these changes may also cause additional events to be placed on the event list. The process of removing the first node from the even list and effecting the changes that it causes is repeated until the event list is empty.

When an arrival node is removed from the event list, a node representing the arriving customer is placed on the shortest of the four teller queues.

If that customer is the only one on a queue, a node representing his or her departure is also placed on the event list, since he or she is at the front of the queue. At the same time, the next input line is read and an arrival node representing the next customer to arrive is placed on the event list. There will always be exactly one arrival node on the event list, since as soon as one arrival node is removed from the event list another is added to it.

When a departure node is removed from the event list, the node representing the departing customer is removed from the front of one of the four queues. At that point the amount of time that the departing customer has spent in the bank is computed and added to a total. At the end of the simulation, this total will be divided by the number of customers to yield the average time spent by a customer. After a customer node has been deleted from the front of its queue, the next customer on the queue becomes the one being serviced by that teller and a departure node for that next customer is added to the event list.

This process continues until the event list is empty, at which point the average time is computed and printed. Note that the event list itself does not mirror any part of the real world situation. It is used as part of the program to control the entire process. A simulation such as this one, which proceeds by changing the stimulated situation in response to the occurrence of one of several events, is called an event – driven simulation.

Questions:-

1. What is linked list
2. Explain the advantage and disadvantage of using linked list.
3. Explain the insertion process of linked list.
4. Explain with an example how would you delete the first and last node from a list.
5. write a program to create and display the content of list.
6. Explain list implementation of priority queue.
7. Explain in detail array implementation of lists
8. Explain simulation using linked list.
9. Explain simulation process

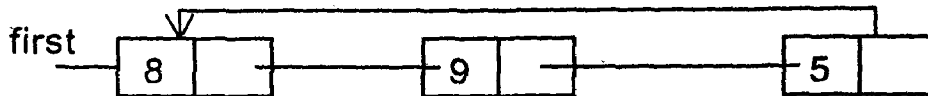
10.7:Other List structures:

Circular list:

It is a list in which every node has a successor, the last element is succeeded by the first element. The advantage of using a circular list with data that is linear in nature is that we can reach both ends of the list using a single external pointer.

Circular Linked List

- It allows to tranverse from the last node to the first node in the list.



Algorithm for inserting an element into circular list:

a. Find the place where the new element belongs check each node in the list until,

1. a key greater than or equal to is encountered
2. end of the list is reached

Create space for the new element

Allocate space for new node using malloc function.

put new element in the list.

The new code can be put in 3 different position

* Algorithm for inserting at beginning of the list

```
/* normal insertion process */
newnode → next = first
first = newnode

/* In addition the last node next points to the first node */
rear → next = newnode
```

* Algorithm for inserting at end of the list

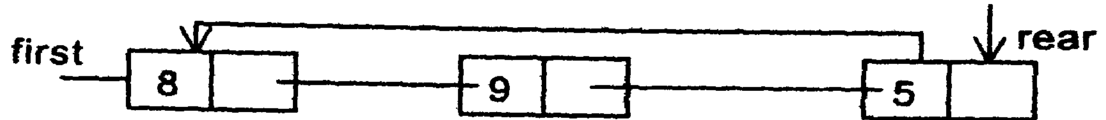
```
/* normal insertion process */
1. newnode → next = rear → next
2. rear →next = newnode
3. rear = newnode
```

* put new element into empty list

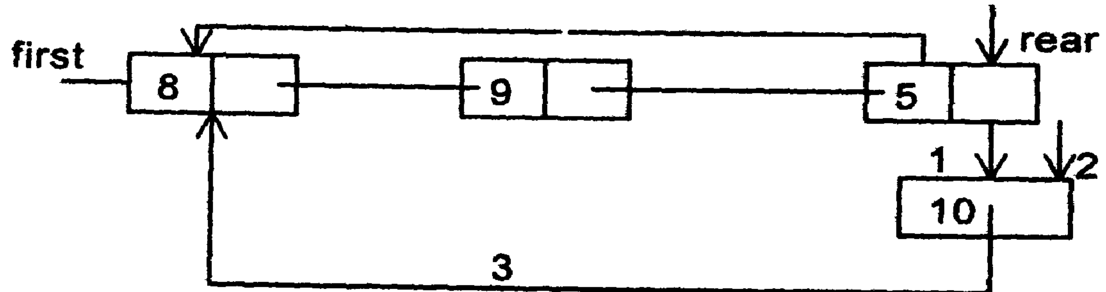
```
first = newnode;
newnode → next = newnode;
```

Addition to circular List

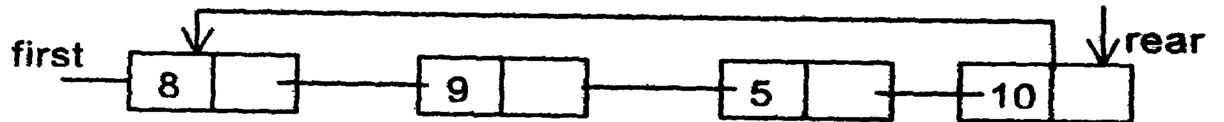
Before insertion:



Insertion process:



After insertion:



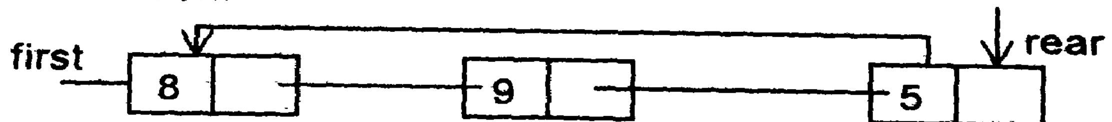
Deletion:

Algorithm to delete the first node from the list is given below:

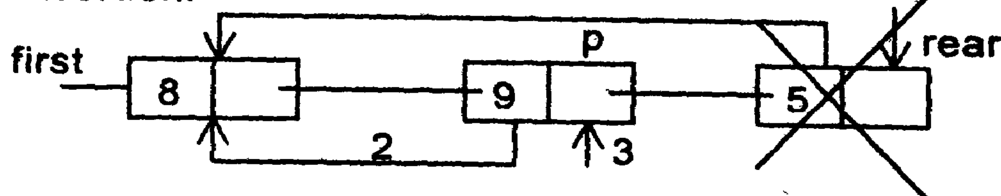
first = first → next
rear → next = first

Deletion from circular list

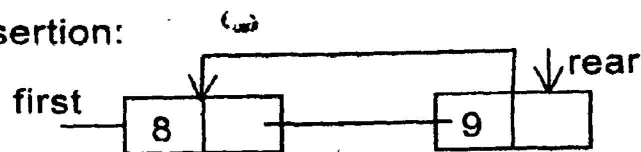
Before insertion:



Deletion insertion:



After insertion:



Recursive operation on list:

The following operations are implemented using recursive function

* To find out the length of the given list
* To copy the source list into target list
* To delete the given list
* To display the content of the given list

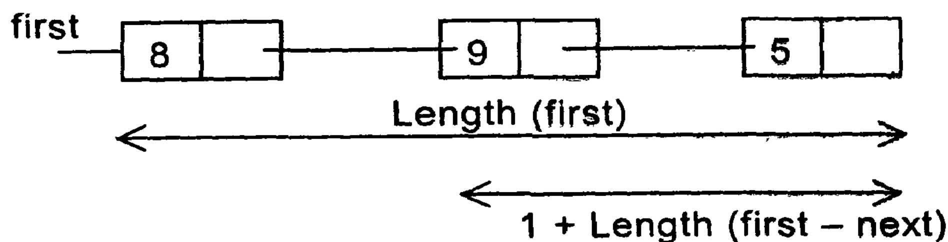
Recursive Length operation on list:

Algorithm for length (node *list)

* Termination condition
  If list is empty return zero
* Recurrence equation (ie)
  If list is not empty
  return (1+length(list → next))

Segment of code:

```
int Length (node * first)
{
    if(first ==null)
        return 0
    else
        return (1+Length(first → next))
}
```



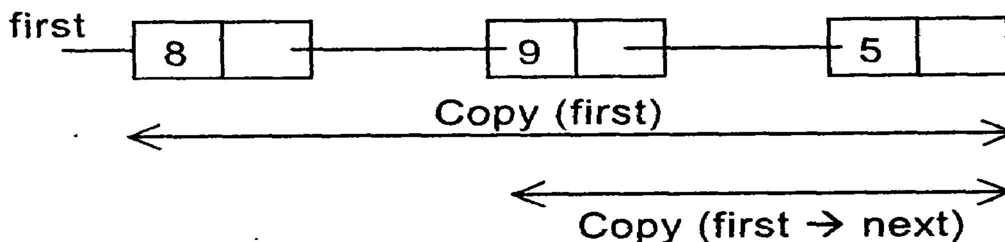
Algorithm for copy (node *list)

* Termination condition
  If list is empty return null
* Recurrence equation(ie)
  If list is not empty
  Step1 : Create a new code

Step2: Copy the value into the data part
Step3: return (copy (list → next))

Segment of code:

```
node *copy (node *first0
{
    if(first == null)
      return 0
    else
    {
        nn = create a node
        nn → data = first → data;
        nn → next = copy(first → next);
    }
}
```

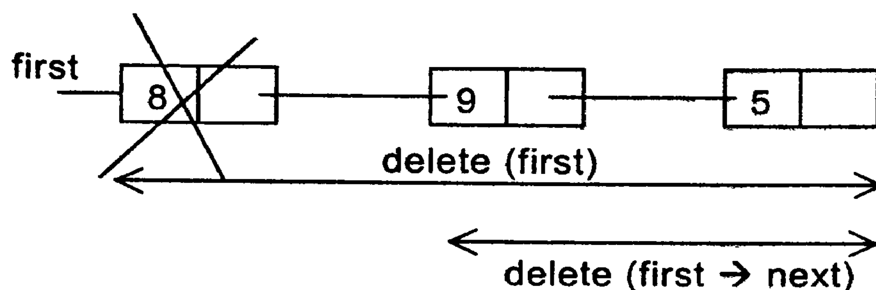


Algorithm for delete (node *list)

- Termination condition
  If list is empty return
- Recurrence equation(ie)
  If list is not empty
  delete the first node in the list
  delete (list → next)

Segment of code:

```
Void Delete (node *first)
{
    if(first == null)
      return;
    else
    {
       first = first → next;
       delete(first);
    }
}
```



Questions:

1. Give brief note on circular linked list.
2. Write a program to convert the linear linked list into circular list.
3. Explain with an example any two operation on circular list.
4. Write a program to recursively implement any two operations on the list.

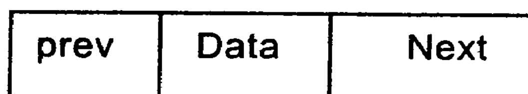
Double linked list:

The main drawback of single linear linked list is that we cannot traverse the list from last node to first node. To avoid this we can store in each node not only the address of next node but also the address of the previous node in the linked list.

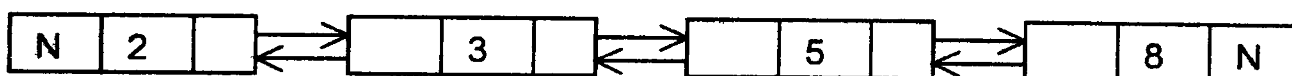
Each node of doubly linked list has two address fields, one named next to move in the forward direction and one named prev to move in the backward direction. Thus a node in doubly linked list has atleast 3 fields.

* Information field : which contains the user's data
* Next fieldsi contains pointer to the next node in the list.
* prev fields : contains pointer to the previous node in the list.

Node structure of Doubly linked list:



Doubly linked list

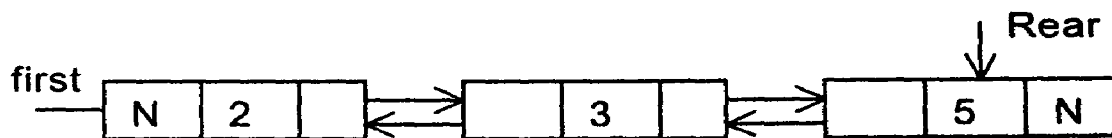


Insertion :

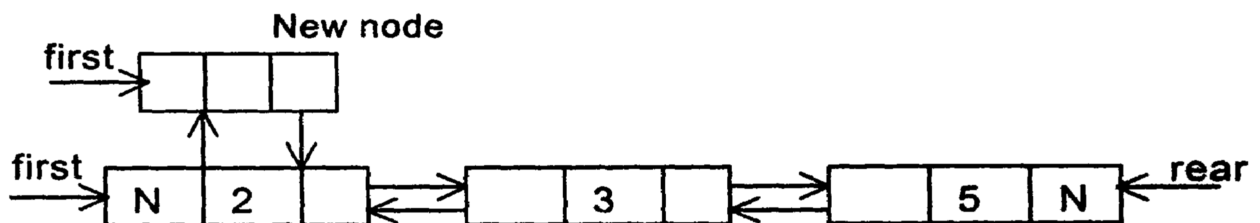
1. Find the place where the new element belongs check each node in the list until
   * A key greater than or equal to is encountered
   * End of the list is reached.

2. Create space for the new element
   Allocate space for new node using malloc function.
3. Put new element in the list
   The new code can be put in 3 different position.
   * Algorithm for inserting at beginning of the list
     /* normal insertion process */
     new node → next = first
     new node → prev = Null
     first → prev = new node
     first = new node

Before Insertion:



Insertion process:

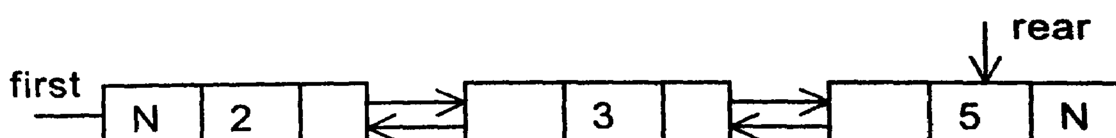


After insertion:

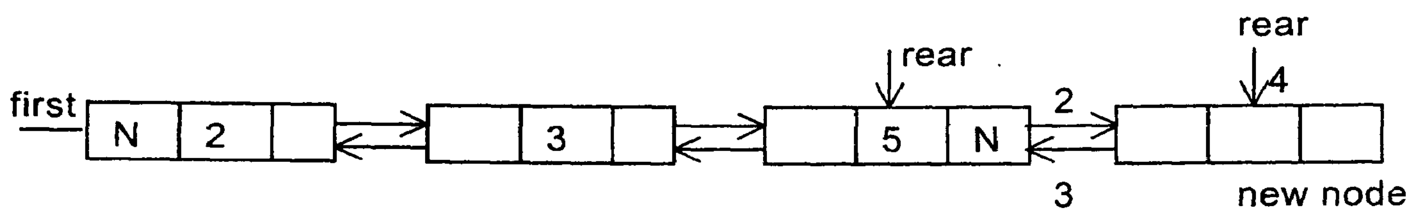
   * Algorithm for inserting at end of the list
     /* Normal insertion process */
     1. new node → next = Null
     2. rear → next = newnode
     3. newnode → prev = rear
     4. rear = new node

Insertion operation

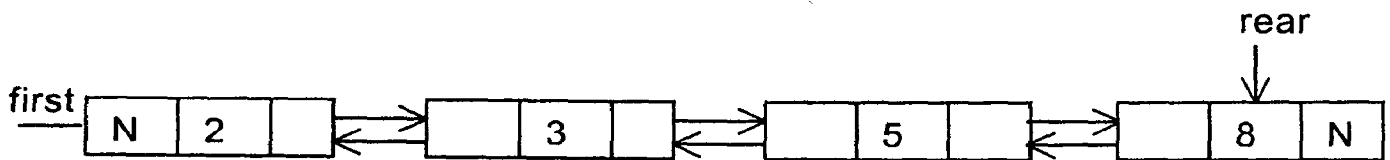
Before insertion:



Insertion process:



After insertion:



* Put new element into empty list

  New node → next = newnode → prev = NULL

  first = rear = newnode;

Deletion:

Here again we have two situation

* Deleting the first node
* Deleting the last node

In the first case, the head pointer is altered to point to the second node in the list. In other cases, the pointer of the node following the deleted node.

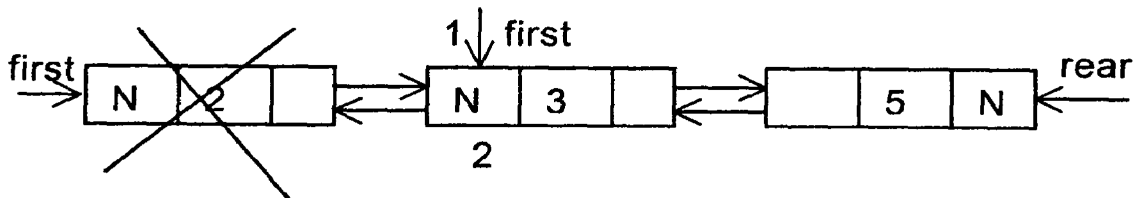
Algorithm to delete the first node from the list is given below.

1. Make the first node's next point the second node in the list.
2. Put N in the second node's previous.

Segment of code:

first = first → next

first → prev = null.

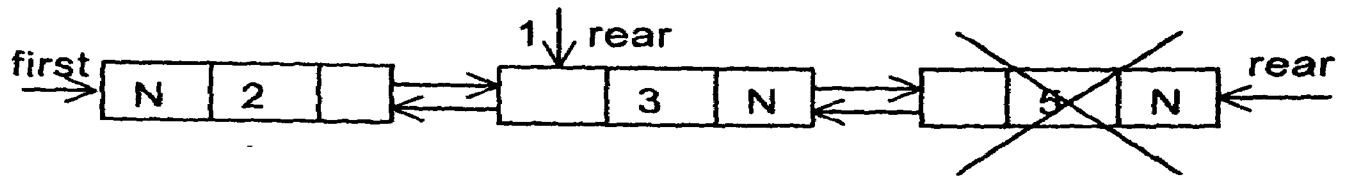


Algorithm to delete the last node from the list is given below:

1. Make rear point the node previous to it (p)
2. Put N in the P's next

Segment of code :

```
rear = rear → prev
rear → next = null
```



Questions:

1. Write a program to display the content of the doubly linked list.
2. Explain any two operations of doubly linked list.
3. Explain in detail the insertion process of doubly linked list
4. Given a brief note on doubly linked list.