PROBLEM SOLVING USING C

Dr.Mohamed Basheer K.P

PROBLEM SOLVING USING C

e- book for BSc. Computer Science Complimentary Students

University of Calicut

Dr. K.P Mohamed Basheer

UNIT I

INTRODUCTION

Introduction

Computers are now affecting every sphere of human activity and bringing about many changes in industry, government, education, medicine, scientific research, law, social sciences, arts, etc. The areas of application of computers are confined only by limitations of human creativity and imagination. In fact any task that can be carried out systematically, using a precise step-by-step method, can be performed by a computer. However, the computer cannot do anything on its own. It must be instructed to do a desired job. Hence, it is necessary to specify a sequence of instructions that a computer must perform to solve a problem. Such a sequence of instructions written in a language that can be understood by a computer is called a computer program. A computer programmer cannot write the instructions to be followed by a computer unless the programmer knows how to solve the problem manually. Thus, before writing a computer program, the programmer must be clear about the processing steps to be performed by the computer. To produce an effective computer program, the programmer must first plan the logic (the various steps) of the program. The step-by-step procedure can be planned either in the form of precise steps, called Algorithm, or in the form of picture showing the logical sequence operations, called Flowchart.

Algorithm

The term algorithm may be formally defined as a sequence of precise and unambiguous instructions for solving a problem in a finite number of operations. An algorithm must possess the following characteristics.

- Each and every instruction should be precise and unambiguous. In other words, each instruction must be definite, meaning that it must be perfectly clear what should be done.
- Each instruction should be effective, i.e., each instruction should be such that it can be performed in a finite time.
- One or more instructions should not be repeated infinitely. This ensures that the algorithm will ultimately terminate.
- An algorithm may have zero or more inputs, which are externally supplied.
- An algorithm must produce the desired output(s).

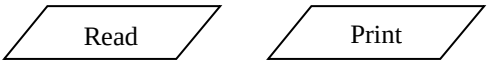
Flowchart

Flowchart is the pictorial representation of a data processing procedure. It makes use of geometrical symbols to denote different types of instructions. The actual instructions are written within these symbols using clear and concise statements. These symbols are connected by solid lines having arrow marks to indicate the flow of operation, i.e., the exact sequence in which the instructions are to be executed. For easy visual recognition, a standard convention is used in drawing flowcharts. In this standard convention, the following shapes are frequently used.

Terminal

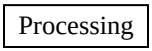
Terminal symbol is used to indicate the beginning (Start), ending (Stop) and pauses (Halt) in the program logic flow.

Input/Output



The input/output symbol is used to denote any function of an input/output device in the program.

Processing

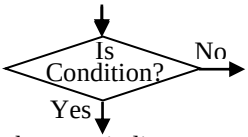


A processing symbol is used in a flowchart to represent arithmetic and data movement operations.

Flow lines

Flow lines with arrowheads are used to indicate the flow of operations.

Decisions



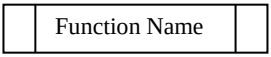
A decision symbol is used in flowchart to indicate a point at which a decision has to be made and a branch to one or more alternative points is possible.

Connector

Connector symbols represent an entry from or an exit to another part of the flowchart. A connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link.

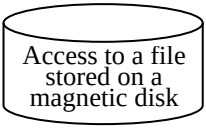
Predefined Process

The predefined process symbol is used to show the execution of a function that is defined elsewhere in the flowchart or on a separate flowchart. It is drawn buy adding two vertical lines to the process symbol, as shown below.



Magnetic Disk

The access to data file stored on a magnetic disk is represented by the following disk symbol



Sample Flowcharts and Algorithms

Example 1:

Write algorithm and draw flowchart to check whether the given three sides can form a triangle. If so display its nature and area.

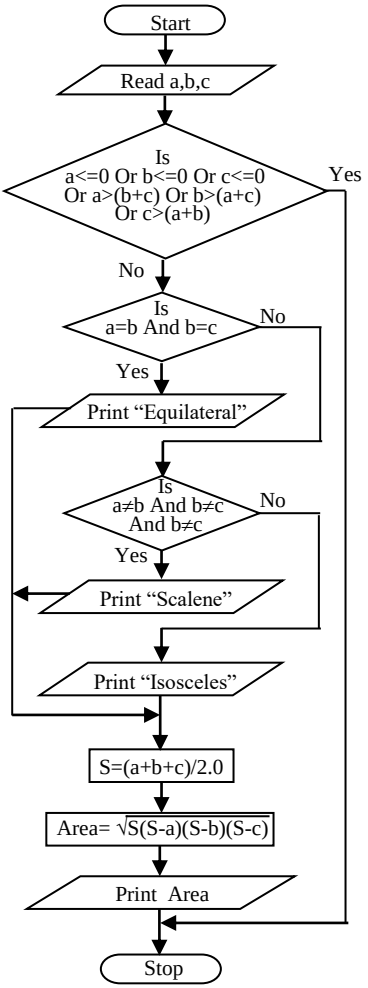
Algorithm

Step 1: Start.
Step 2: Read the sides a, b and c.
Step 3: If a<=0 Or b<=0 Or c<=0 Or
a>(b+c) Or b>(a+c) Or c>(a+b)
Then
Print “Cannot from a Triangle”
Go to Step 8.
Step 4: If a=b And b=c
Then
Print “Equilateral Triangle”
Else
If a≠b And a≠c And b≠c

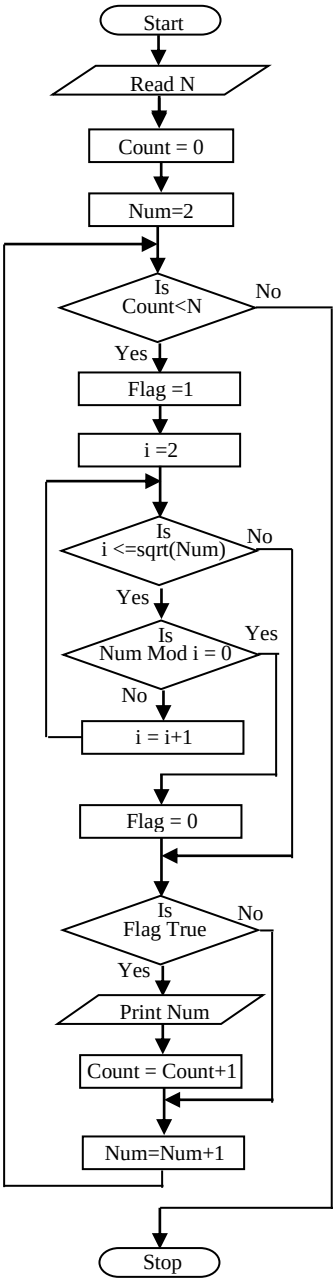
Then
Print “Scalene Triangle”
Else
Print “Isosceles Triangle”.

Step 5: s←(a+b+c)/2.
Step 6: Area←Sqrt(s(s-a)(s-b)(s-c)).
Step 7: Print Area.
Step 8: Stop.

Flowchart of Example 1



Flowchart of Example 2



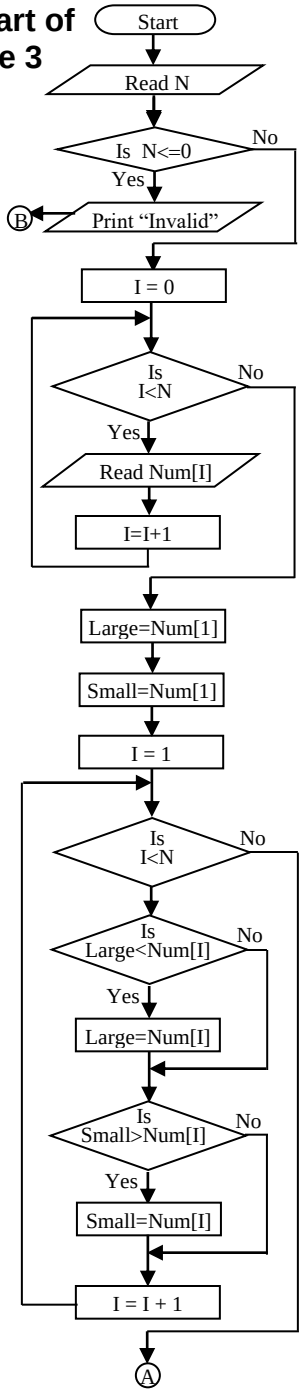
Example 2:

Write algorithm and draw flowchart to display the first 'n' prime numbers.

Algorithm

Step 1: Start
Step 2: Read N, the number of prime numbers to be displayed.
Step 3: Set Count←–0 and Num←–2
Step 4: Repeat steps 5 through 11 until Count<=N
Step 5: Set Flag←–1
Step 6: Set i←–2
Step 7: Repeat steps 8 through 9 until i<=Sqrt(N)
Step 8: If Num Mod I = 0
Then
Flag←–0
Go to Step 10
Step 9: i←–i+1
Step 10: If Flag = 1
Then
Print Num
Count←–Count+1
Step 11: Num←–Num+1
Step 12: Stop.

Flowchart of Example 3



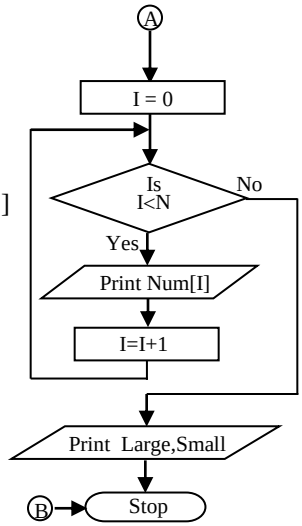
Example 3:

Write algorithm and draw flowchart to find the largest and smallest among 'n' numbers. Display the numbers entered along with the largest and smallest.

Algorithm

Step 1: Read N, the number of numbers to be entered.
Step 2: If N<=0
Then
Print "Invalid Data"

Go to Step 18.
Step 3: Set I←0
Step 4 Repeat Steps 5 through 6 until I<N
Step 5: Read Num[I], the next number.
Step 6: I←I+1
Step 7: Set Large←Num[1] and Small←Num[1]
Step 8: Set I←1
Step 9: Repeat Steps 10 through 12 until I<N
Step 10: If Large < Num[I]
Then
Large←Num[I]
Step 11: If Small > Num[I]
Then
Small←Num[I]
Step 12: I←I+1
Step 13: Set I←0
Step 14: Repeat Steps 15 through 16 until I<N
Step 15: Print Num[I], the next number.
Step 16: I←I+1
Step 17: Print Large and Small
Step 18: Stop.



Example 4:

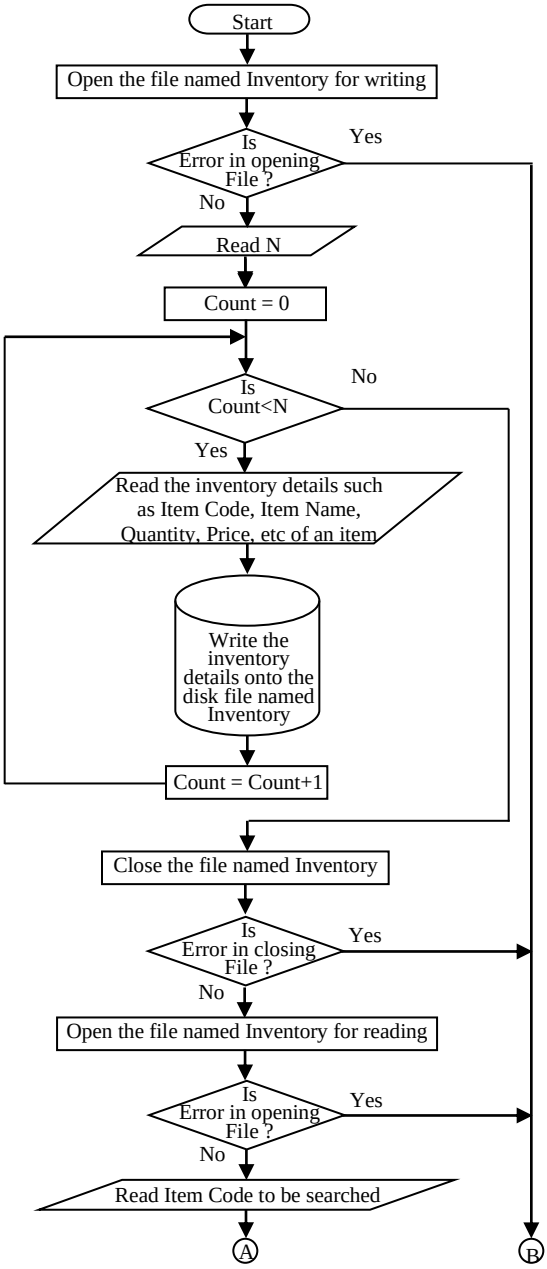
Write an algorithm to multiply two matrices. Display the matrices entered along with the product matrix.

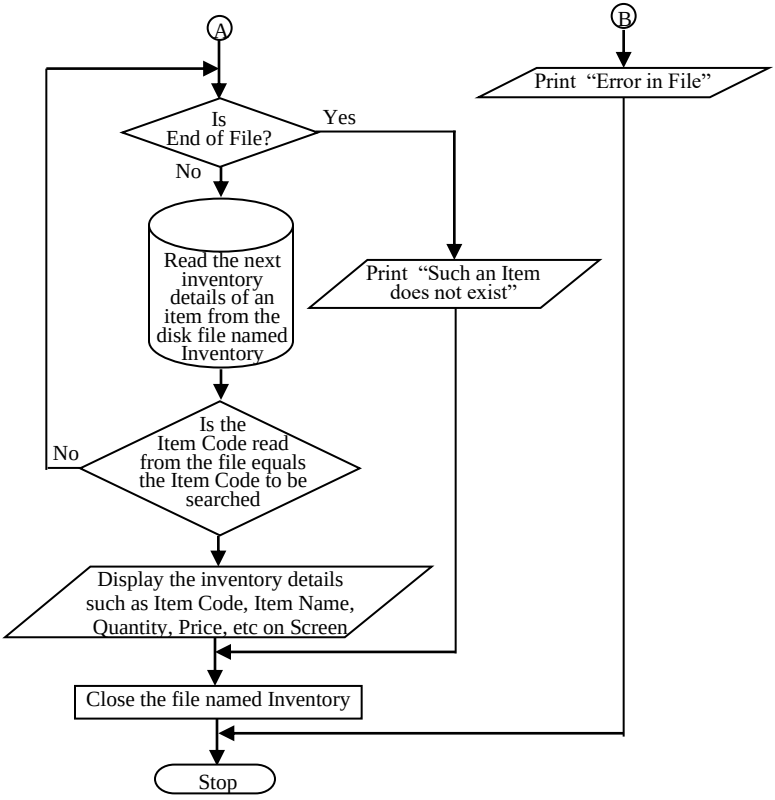
Step 01:Start
Step 02:Read Row1 and Col1, the order of the first matrix.
Step 03:Read Row2 and Col2, the order of the second matrix.
Step 04:If Col1≠Row2
Then
Print "Order mismatch! Multiplication not possible!"
Go to Step 49.
Step 05:Set I←0
Step 06:Repeat Steps 7 through 11 until I<Row1
Step 07:Set J←0
Step 08:Repeat Steps 9 through 10 until J<Col1
Step 09:Read Mat1[I][J], the next element of the first matrix.
Step 10:J←J+1
Step 11:I←I+1
Step 12:Set I←0
Step 13:Repeat Steps 14 through 18 until I<Row2
Step 14:Set J←0
Step 15:Repeat Steps 16 through 17 until J<Col2

Step 16:Read Mat2[I][J], the next element of the second matrix.
Step 17:J←J+1
Step 18:I←I+1
Step 19:Set I←0
Step 20:Repeat Steps 21 through 28 until I<Row1
Step 21:Set J←0
Step 22:Repeat Steps 23 through 27 until J<Col2
Step 23:Set K←0
Step 24:Repeat Steps 25 through 26 until K<Col1
Step 25:Mat3[I][J]=Mat3[I][J]+Mat1[I][K]×Mat2[K][J]
Step 26: K←K+1
Step 27:J←J+1
Step 28:I←I+1
Step 29:Set I←0
Step 30:Repeat Steps 31 through 35 until I<Row1
Step 31:Set J←0
Step 32:Repeat Steps 33 through 34 until J<Col1
Step 33:Print Mat1[I][J], the next element of the first matrix.
Step 34:J←J+1
Step 35:I←I+1
Step 36:Set I←0
Step 37:Repeat Steps 38 through 42 until I<Row2
Step 38:Set J←0
Step 39:Repeat Steps 40 through 41 until J<Col2
Step 40:Print Mat2[I][J], the next element of the second matrix.
Step 41:J←J+1
Step 42:I←I+1
Step 43:Set I←0
Step 44:Repeat Steps 45 through 49 until I<Row1
Step 45:Set J←0
Step 46:Repeat Steps 47 through 48 until J<Col2
Step 47:Print Mat3[I][J], the next element of the product matrix.
Step 48:J←J+1
Step 49:I←I+1
Step 49:Stop.

Example 5:

Draw a flowchart to create an invetory file. Searh an item in the file and display its details, if the item exists in the file.





Features of a Good Program

The following are the desirable characteristics of a computer program.

Integrity

This refers to the accuracy of the calculations. The integrity of the calculations is an absolute necessity in any computer program.

Clarity

This refers to the overall readability of the program, with particular emphasis on its underlying logic.

Simplicity

The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with overall program objectives. In fact, it may be desirable to sacrifices a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.

Efficiency

It is concerned with the execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity.

Modularity

Main program can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as separate program module. The use of modular programming structure enhances the accuracy, clarity and efficiency of a program, and it facilitates future program alterations. (The concept of modularisation will be discussed later in Unit XI)

Generality

The program should be as general as possible, within reasonable limits.

Questions

3 Mark Questions

1. What is an algorithm? List the necessary characteristics of an algorithm.
2. What is a flowchart? List the commonly used flowchart symbols and their function.
3. List and explain the desirable characteristics of a good program.
4. What is meant by Modular programming? Explain its advantages.

UNIT II

C
FUNDAMENTALS

2.1 Introduction to C

C is general purpose, structured programming language. Dennis Ritchie at the Bell Laboratories in USA originally developed C in the 1970s. C is the result of the development process that started with an older language called BCPL (Basic Combined Programming Language) developed by Martin Richards, and it influenced a language called B, which was invented by Ken Thompson. B led to the development of C in 1970s.

Since those early days, C has undergone several changes. Existing features have been modified, new features have been added and some obsolete ones deleted. With the advent and proliferation of microcomputers, several implementations of C emerged. Though compatible to a great degree, there were discrepancies and anomalies in these implementations. In 1983, the American National Standard Institute (ANSI) established a committee to define a standard version of C language. The ANSI standard was finally adopted in December 1989. Major producers of C compilers have adopted this standard

C became the most popular language of that day. One of the reasons for its wide spread popularity was its flexibility - it allows one to program in a 'standard' way yet it permits great 'freedom of expression'. It combines the control structures normally found in high-level languages such as Pascal or Ada with the ability to manipulate bits, bytes and addresses, something usually associated with assembly language. This flexibility allows C to be used for system programming (e.g., for writing operating systems) as well as for application programming. Since the C language combines the best elements of high-level languages with the control and flexibility of assembly languages, it is often referred to as middle-level language.

C is characterized by the ability to write very concise source program, partly due to the large number of operators included within the C language. C lends itself to modular programming. It has a relatively small instruction set, though the actual implementations include extensive library functions, which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the users can easily extend the features and capabilities of the language.

C compilers and interpreters are commonly available for computers of all sizes. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages.

Another important characteristic of C is that its programs are highly portable. The reason for this is that C relegates most computer dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for particular characteristics of the host computer. These library functions are relatively

standardized. Therefore, most C programs can be processed on many different computers with no or little alteration.

2.2. Structure of the 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. Any other function definition must be defined separately, either ahead or after main. Each function must contain:

1. A function heading, which consists of the function name, followed by an optional list of arguments, enclosed in parentheses.
2. A list of argument declarations, if arguments are included in the heading.
3. A compound statement, which comprises the remainder of the function.

Each compound statement is enclosed within a pair of braces, i.e., {}. These braces may contain one or more elementary statements (called expression statements) and other compound statements. Each expression statement must end with a semicolon (;).

Comments (remarks) may appear anywhere within a program, as long as they are placed in within the delimiters, /* and */. Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

Here is an elementary C program, which illustrates the overall program organization.

```
    /* Program to calculate                  /* Title comment */
       the area of a rectangle */
#include<stdio.h>                            /* Library file name */
main()                                       /* Function heading */
{
        float length,breadth,area;           /* Variable declaration */
        clrscr();                            /* Library function call */
        printf("Enter the length : ");       /* Output prompt message */
        scanf("%f",&length);                 /* Input statement */
        printf("Enter the breadth : ");      /* Output prompt message */
        scanf("%f",&breadth);                /* Input statement */
        area = length*breadth;               /* Assignment statement */
        printf("Area = %f",area);            /* Output statement */
}
```

2.3. The C Character Set

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form to basic program elements (e.g., constants, variables, operators, expressions, etc.). The special characters are listed below.

+	-	*	/	=	&	#	!
?	^	”	’	~	\	|	<
>	(	)	[	]	{	}	:
;	.	_	%	Blank Space			

C uses certain combinations of these characters, such as \b, \n, \t, <=, >=, &&, ||, !=, etc., to represent special conditions.

2.4. Identifiers

Identifiers are the names that are given to various program elements, such as variables, functions and arrays. The rules of forming an identifier are:

1. Identifier must begin with a letter of alphabet. In C, the underscore (_) character is considered a letter.
2. The first character may be followed by a sequence of letters and/or digits (0 through 9).
3. An identifier can be arbitrarily long. However some implementations of C recognizes only the first 8 characters.
4. Both upper- and lower-case characters are permitted. However, they are not interchangeable, i.e., an uppercase letter is not equivalent to the corresponding lowercase letter.
5. No identifier may be a keyword.
6. No special characters, such as blank space, period, semicolon, comma, or slash, are permitted.

The following names are valid identifiers.

Count total sum_1 temperature tax_rate TABLE

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

4th	The first character must be a letter of alphabet
order-no	Illegal character (-)
error flag	Illegal character (blank space)
int	Keyword

2.5. Keywords

Keywords are the standard identifiers that have standard, predefined meaning in C. These keywords can be used only for their intended purpose and they cannot be used as programmer-defined identifiers. The standard keywords are:

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

Note that keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier, but it is not a good programming practice.

2.6. Data Types

The C language defines five fundamental data types: character, integer, floating-point, double floating-point and valueless. These are declared using the keywords char, int, float, double and void, respectively. These types form the basis for several other types. The size and the range of these data types may vary among processor types and compilers. Typical memory requirements of the basic data types are given below.

Data type	Meaning	Size (byte)	Minimal Range
char	Character	1	-128 to 127
int	Integer	2	-32,768 to 32767
float	Single precision real number	4	3.4E-38 to 3.4E+38 with 6 digits of precision
double	Double precision real number	8	1.7E-308 to 1.7E+308 with 10 digits of precision
void	Valueless	0	

Void is used to specify an empty set of values

2.7. Data Type Qualifiers (Type Modifiers)

Except type void, the basic data types may have various modifiers or qualifiers preceding them. A data type qualifier alters the meaning of the base type to more precisely fit a specific need. The commonly used data type qualifiers are signed, unsigned, short and long. The int base type can be modified by signed, unsigned, short and long. The char can be modified by signed and unsigned. One may also apply long to double.

The interpretation of qualified integer data type will vary from one C compiler to another, though there are some commonsense relationships. Thus, a short int may require less memory than an ordinary int or it may require the same amount of memory as an ordinary int, but it will never exceed an ordinary int in word length. Similarly, a long int may require the same amount of memory as an ordinary int or it may require more memory, but it will never be less than the ordinary int.

If short int and int both have the same memory requirements (e.g., 2 bytes) then long int will generally have double the requirements (e.g., 4 bytes) or if int and long int both have the same memory requirements (e.g., 4 bytes) then short int will generally have half the memory requirements (e.g., 2 bytes).

The use of signed on integers is allowed, but it is redundant because the default integer declaration assumes a signed number. The most important use of signed is to modify char in implementations in which char is unsigned by default.

An unsigned int has the same memory requirement as an ordinary int. However, in the case of an ordinary int (or a short int or long int), the left most bit is reserved for the sign. With an unsigned int, all of the bits are used to represent the numerical value. Thus, a signed int can be approximately twice as large as an ordinary int (though, of course, negative values are not permitted). For example, if an ordinary int can vary form –32,768 to +32767 (which is typical for 2-byte int), then an unsigned int will be allowed to vary from 0 to 65,535. The unsigned qualifier can also be applied to other qualified integers, e.g., unsigned short int or unsigned long int.

Some compilers permit the qualifier long to be applied to float or double, e.g., long float, or long double. However, the meaning of these data types will vary from one C compiler to another. Thus long float may be equivalent to double. Moreover, long double may be equivalent to double, or it may refer to a separate extra-large double precision data type requiring more than 8 bytes of memory.

When a type quantifier is used by itself (i.e., when it does not precede a basic type), then int is assumed. For example, the type specifies

signed, unsigned, short and long are same as signed int, unsigned int, short int and long int, respectively.

The following table shows all valid data type combinations supported by C, along with their minimal ranges and typical memory size.

Type	Memory Size (bytes)	Minimal Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to +127
int	2 or 4	-32,768 to 32,767
unsigned int	2 or 4	0 to 65,535
signed int	2 or 4	Same as int
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
signed short int	2	Same as short int
long int	4	-2,147,483,647to 2,147,483,647
signed long int	4	Same as long int
unsigned long int	4	0 to 4,294,967,295
float	4	1E-38 to 1E+38 with 6 digits of precision
double	8	1E-308 to 1E+308 with 10 digits of precision
long double	8	1E-308 to 1E+308 with 10 digits of precision

2.8. Constants

The term constant means that it does not change during the execution of a program. There are four basic types of constants in C. They are:

1. Integer constants
2. Floating-point constants
3. Character constants
4. String constants

Integer and floating-point constants represent numbers. They are often referred to collectively as numeric-type constants. The following rule applies to all numeric type constants:

- Comma and blank spaces cannot be included within the constants.
- Constants can be preceded by a – or + sign, if desired. If either sign does not precede the constant it is assumed to be positive.

- The value of a constant cannot exceed specified minimum and maximum bounds. For each type of constant, these bounds vary from one C compiler to another.

Integer Constant

Integer constants are whole numbers without any fractional part. Thus integer constants consist of a sequence of digits. Integer constants can be written in three different number systems: Decimal, Octal and Hexadecimal.

A decimal integer constant can consist of any combination of digits taken from the set 0 through 9. If the decimal constant contains two or more digits, the first digit must be something other than 0.

The following are valid decimal integer constants

0 1 1234 743 -8321

The following decimal integer constants are written incorrectly for the reasons stated.

12,345	Illegal character (,).
132.0	Illegal decimal point (.).
10 20 30	Illegal character (blank space).
098	First digit cannot be zero.

An octal integer constant can consist any combination of digits taken from the set 0 through 7. However, the first digit must be 0, in order to identify the constant as an octal number.

The following are valid octal integer constants.

0 01 0743 -0743 +04232

The following octal integer constants are written incorrectly for the reasons stated.

743	Does not begin with 0
05283	Illegal digit (8)
07.32	Illegal character (.)

A hexadecimal integer constant must begin with either 0x or 0X. It can then be followed by any combination of digits taken from the sets 0 through 9 and A through F (either upper- or lower-case).

The following are valid hexadecimal integer constants

0X0 0x1 0X7FAB 0xabcd -0xface

The following hexadecimal integer constants are written incorrectly for the reasons stated.

0x12.34	Illegal character (.)
0ABCDE	Does not begin with ox.
oxagfedc	Illegal character (g)

Qualified Integer Constants

The data type qualifier short, long, signed and unsigned may be applied to any integer. Unsigned integers are positive. It can be used to increase the range of positive numbers normally stored. Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of 2, though they may not be negative. An unsigned integer constant can be identified by appending the letter U (either upper- or lower case) to the end of the constant.

Long integer constant may exceed the magnitude of ordinary integer constants, but require more memory within the computer. A long integer constant can be identified by appending L (either upper- or lower case) to the end of the constant.

An unsigned long integer may be specified by appending UL (either upper- or lower case) to the end of the constant.

Examples:

Data	Description
2468	Decimal integer
-246878325L	Decimal long integer
2468U	Decimal unsigned integer
24687835UL	Decimal unsigned long integer
0123456L	Octal long integer
07777U	Octal unsigned integer
01234567UL	Octal unsigned long integer
0X5000U	Hexadecimal unsigned integer
0XFFFFFUL	Hexadecimal unsigned long integer

Floating-Point Constant

A floating-point constant is a base-10 number that contains either a decimal point or an exponent or both. A floating-point constant can be written in two forms: Fractional form or Exponential form. A floating-point constant in a fractional form must have at least one digit each to the left and right of the decimal point. A floating-point in exponent form consists of a mantissa and an exponent. The mantissa itself is represented as a decimal integer constant or a decimal floating-point constant in fractional form. The mantissa is followed by the letter E or e and the exponent. The exponent

must be a decimal integer. The actual number of digits in the mantissa and the exponent depends on the computer being used.

The following are valid floating-point constants

1.0	0.2	872.602	5000.0
0.000743	315.0066	2E-8	0.006e-3
1.666E+8	.121212e12	-0.156e-4	0.01540e05

The following are some invalid floating-point constants.

1	No decimal point or exponent
1,00.0	Illegal character (,)
2E+10.2	Exponent must be an integer
3 E10	Illegal character (space)

Character Constants

A character constant is a single character, enclosed in single quotation marks.

e.g., 'A' 'X' '3' ' '

Characters are stored internally in computer as coded set of binary digits, which have positive decimal integer equivalents. The value of a character constant is the numeric value of the character in the machine's character set. This means that the value of a character constant can vary from one machine to the next, depending on the character set being used on the particular machine. For example, on ASCII machine the value of 'A' is 65 and on EBCDIC machine it is 193.

Escape Sequences

Certain non-printing characters, as well as backslash (\) and apostrophe ('), can be expressed in terms of escape sequences. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, the newline character (line feed) can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters. The commonly used escape sequences are:

Character	Escape Sequence
Bell (Alert)	\a
Backspace	\b
Horizontal tab	\t
Vertical Tab	\v

Newline	\n
Carriage return	\r
Form feed	\f
Quotation mark (”)	\"
Apostrophe (’)	\'
Question mark (?)	\?
Backslash	\\
Null	\0

The following are some character constants, expressed in terms of escape sequences.

‘\n’ ‘\t’ ‘\b’ ‘\”’ ‘\\’ ‘\0’ ‘\’’

Of particular interest is the escape sequence \0. This represents the null character (ASCII 000), which is used to indicate the end of string. Note that the character constant '0' is different from '\0'

String Constant

A string constant consists of a set of zero or more characters enclosed in quotation marks.

Several string constants are given below.

“green”	“Welcome to C programming”	“”
“ ”	“The largest number is”	“2*(i+3)/j”
“ ”	“Line no.1\nLine no.2\nLine no.3”	“Error\a\a\a”

Note that the string constant “Line no.1\nLine no.2\nLine no.3” extends over three lines, because of the newline characters that are embedded within the string. Thus, the string would be displayed as

Line no.1
Line no.2
Line no.3

Sometimes some special characters (e.g., backslash or quotation mark) must be included as a part a string constant. These characters must be represented in terms of their escape sequences. Similarly, certain nonprinting characters (e.g., tab, newline) can be included in a string constant if they are represented in terms of their corresponding escape sequences.

The following string constant includes three special characters that are represented by their corresponding escape sequences.

“\t To continue, press the \”Return\” key …\n”

The special characters are \t (horizontal tab), \” (double quotation marks, which appears twice), and \n (newline).

The compiler automatically places a null character (\0) at the end of every string constant, as the last character within the string (before closing

the double quotation mark). This character is not visible when the string is displayed. However, we can easily examine the individual characters within a string, and test to see whether or not each character is null character. Thus, the end of every string can be easily identified. This is very helpful if the string is scanned on a character-by-character basis, as is required in many applications. Also, in many situations this end-of-string designation eliminates the need to specify a maximum string length.

Remember that a character constant (e.g., 'A') and the corresponding single character string ("A") are not equivalent. The character constant has an equivalent integer value, whereas a singe-character string constant does not have an equivalent integer value, and in fact, consist of two characters – the specified character followed by the null character (\0).
For example, the character constant 'A' has an integer value of 64 in ASCII character set. It does not have a null character at the end. In contrast, the string constant "A" actually consists of two characters – the upper case letter A and the null character \0. This constant does not have a corresponding integer value.

2.9. Variables

A variable is named location in memory that is used to hold a value that can be modified by the program. Thus variables are the identifiers that are used to represent some specified type of information (data items) within designated portions of the program. Each variable has a specified storage location in memory where its data item is stored. The variable is given a name and the variable name is the 'name tag' for the storage location. The value of the variable at any instant during the execution of program is equal to the data item stored in the storage location identified by the name of the variable. The data item must be assigned to variable at some point in the program. The data item can be accessed later in the program simply by referring to the variable name.

A given variable can assign different data items at various places within the program. Thus, the information represented by the variable can change during the execution of the program. However, the data type associated with the variable cannot change.

Declaring Variables

All variables must be declared before they can appear in executable statements. A declaration associates a group of variables with a specific data type. When a variable name is declared, then a memory location is identified and given this name.

A declaration consists of a data type, followed by one or more variable names separated by commas and ending with a semicolon. Its general format is

type-name variable-name1, variable-name2,…,variable-namen;

For example, a C program contains the following type declarations.

```
int a,b,c;
float root1,root2;
char c;
```

Thus, a, b and c are declared to be integer variables, root1 and root2 are floating-point variables and c is a character type variable.

Variables can be distributed among declarations in any fashion. For example, the above type declarations could also have been written as follows.

```
int a;
int b;
int c;
float root1;
float root2;
char c;
```

This form may be useful if each variable is to be accompanied by a comment explaining its purpose. In small programs, however, items of the same type are usually combined in a single declaration.

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 and long int or simply short and long, respectively.

For example, a C program contains the following type declarations.

```
short int a,b,c;
long int r,s,t;
int p,q;
```

Some compilers will allocate less storage space to short integer variables a, b and c than to integer variables p and q. Typical values are two bytes for each short integer variable and four bytes for each ordinary integer variable. The maximum permissible values of a, b and c will be smaller than the maximum permissible value of p and q when using a compiler of this type. Similarly, some compilers will allocate additional storage space to the long integer variables r, s and t than to the integer variable p and q. Typical values are eight bytes for each long integer variable and four bytes for each ordinary integer variable. The maximum permissible values of r, s and t will be larger than the maximum permissible value of p and q when using a compiler of this type.

The above type declarations could have been written as

```
short  a,b,c;
long r,s,t;
```

```
int p,q;
```

Thus, short and short int are equivalent, as are long and long int.

An integer variable can also be declared to be unsigned, by writing unsigned int, or simply unsigned as the type indicator. Unsigned integer quantities can be larger than ordinary integer quantities, but they cannot be negative.

For example, a C program contains the following type declarations.

```
int a,b;
unsigned x,y;
```

The unsigned variables x and y can represent values that are twice as large as the value represented by a and b. However, x and y cannot represent negative quantities. For example, if the computer uses 2 bytes for each integer quantity, then a and b may take on values that range from –32768 to +32767, whereas the values of x and y may vary from 0 to +65535.

Floating-point variables can be declared to be double precision by using the type indicator double or long float.

For example, a C program contains the following type declarations.

```
float c1,c2,c3;
double root1,root2;
```

The last declaration could have been written as

```
long float root1, root2;
```

Initial values can be assigned to variable within a type declaration. To do so, the declaration must consist of a data type, followed by variable name, an equal sign (=) and a constant of the appropriate type.

For example, a C program consists of the following type declarations.

```
int c = 12;
char star = '*';
float sum = 0.0;
double factor = 0.21023e-6;
```

Thus, c is an integer variable whose initial value is 12, star is a character type variable initially assigned the character '*', sum is the floating-point variable whose initial value is 0.0, and factor is a double-precision variable whose initial value is 0.21023×10^{6}.

2.10. Expression

An expression may consist of single entity (such as a constant or an array element or a variable or a reference to a function) or some combination

of such entities, interconnected by one or more operators. An expression represents a single data item such as a number or character.

Expressions can also represent a logical condition that is either true or false. However, in C, the conditions true and false are represented by the integer values 1 and 0, respectively. Hence logical-type expressions really represent numerical quantities.

Several simple expressions are shown below.

```
a+b     x=y     2==3     x<=y     2*(i+3.8)/(j*2)     ++i
```

2.11. Statements

A statement specifies an action. Thus a statement causes the computer to carry out some action. There are three different classes of statements in C. They are

Expression statements
Compound statements
Control Statements

An expression statement consists of a valid expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Several expression statements are shown below

```
func();   /* a function call */
c=a+b;    /* an assignment statement */
b+f();    /* a valid, but strange statement */
;         /* an empty or null statement */
```

The first expression statement executes a function call. The second is an assignment statement. The third expression, though strange, is still evaluated by the compiler because the function f() may perform some necessary task. The final expression shows that a statement can be empty (sometimes called a null statement

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. Unlike expression statements, compound statement does not end with a semicolon.

A typical compound statement is shown below.

```
{
  pi=3.14;
  circumference= 2.0 * pi * radius;
  area = pi * radius * radius;
}
```

This particular compound statement consists of three assignment-type expression statements.

Control statements are used to create special program features, such as logical tests, loops and branches. Control statements will be discussed in unit X.

2.12. Symbolic Constants

A symbolic constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant, or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. Symbolic constants are usually defined at the beginning of the program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc., that the symbolic constants represent.

A symbolic constant is defined by writing

```
#define name text
```

where name represents a symbolic name, typically written in upper-case letters and text represents the sequence of characters that is associated with the symbolic name. Note that the text does not end with a semicolon, since a symbolic constant definition is not a true C statement.

For example, a C program contains the following symbolic constant definitions.

```
#define TAXRATE 0.23
#define PI 3.141593
#define TRUE 1
#define FALSE 0
#define MESSAGE “Welcome”
```

Notice that the symbolic names are written in upper case, to distinguish them from ordinary C identifiers. Also note that the definitions do not end with semicolons.

Now suppose that the program contains the statement

```
area = PI * radius * radius;
```

During the compilation process, each occurrence of symbolic constant will be replaced by its corresponding text. Thus, the above statement will become

```
area = 3.141593 * radius * radius;
```

Now suppose that the semicolon had been incorrectly included in the definition for PI, i.e.,

```
#define PI 3.141593;
```

The assignment statement for area would then become

```
area = 3.141593; * radius * radius;
```

Note the semicolon preceding the first asterisk. This is clearly incorrect, and causes an error in compilation.

The substitution of the text for a symbolic constant will be carried out anywhere beyond the #define statement, except within a string. Thus any text enclosed in double quotation marks will be unaffected by this substitution process.

For example, a C program contains the following statements.

```
#define CONSTANT 6.023
int c;
…
printf("CONSTANT = %f", c);
```

The printf statement will be unaffected by symbolic constant definition, since the term "CONSTANT = %f" is string constant. If, however, the printf statement were written as

```
printf("CONSTANT = %f", CONSTANT);
```

The printf statement would become

```
printf("CONSTANT = %f", 6.023);
```

during the compilation process.

The use of symbolic constant is recommended when writing C programs, since they contribute to the development of clear, orderly programs. For example, symbolic constants are more readily identified than the information they represent, and the symbolic name usually suggests the significance of their associated data items. Further more, it is easier to change the value of single symbolic constant than to change every occurrence of some numerical constant that may appear in several places within the program.

2.13. Operators In C

C language is very rich in built-in operators. The commonly used operators include

1. Arithmetic operators
2. Unary operators
3. Relational and Equality operators
4. Logical operators
5. Assignment operators
6. Conditional operator.

2.13.1. Arithmetic Operators

There are five arithmetic operators in C. They are

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division (Modulus Operator)

The operands of the arithmetic operators must represent numerical values – integer quantities, floating-point quantities, or characters. The modulus operator requires that both the operands be integers and the second operand be nonzero. Similarly, the division operator requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if the division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

Suppose that a and b are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

Notice that the truncated quotient resulting from the division operation, since both the operands represent integer quantities.

Now suppose that a and b are floating-point variables whose values are 12.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	14.5
a - b	10.5
a * b	25.0
a / b	6.25

Finally suppose that a and b are character variables that represent the characters P and T, respectively. Several arithmetic expressions that make use of these variables are shown below, together with their resulting values.

Expression	Value
a	80
a + b	164
a + b + 5	169
a + b + '5'	217

Note that 'P' is encoded as 80 in decimal, 'T' is encoded as 84, '5' is encoded as 53 in the ASCII character set.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation towards zero; i.e., the resultant will always be smaller in magnitude than the true quotient. The interpretation of the remainder operation is unclear when one of the operand is negative. Most versions of C assign the sign of the first operand to the remainder.

Suppose that that a and b are integer variables whose values are 11 and -3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	8
a - b	14
a * b	-33
a / b	-3
a % b	2

If a had been assigned value of –11 and b had been assigned 3, then the value of a / b would be –3 but the value of a % b would be –2. Similarly, if a and b had both been assigned negative values (-11 and –3, respectively), the value of a / b would be 3 and the value of a % b would be –2.

Operands that differ in type may undergo type conversion before the expression takes on its final value. This is known as implicit conversion. In general, the final result will be expressed in the highest precision possible, consistent with the data types of the operands. The following rules apply to arithmetic operations between two operators with dissimilar data types.

1. If one of the operands is long double, the other will be converted to long double and the result will be in long double.
2. Otherwise, if one of the operands is double, the other will be converted to double and the result will be double.
3. Otherwise, one of the operand is float, the other will be converted to float and the result will be float.
4. Otherwise, if one of the operand is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int.
5. Otherwise, one of the operand is long int and the other is unsigned int, then:

a. If unsigned int can be converted to long int, the unsigned long int operand will be converted as such and the result will be long int.
b. Otherwise, both operands will be converted to unsigned long int and the result will be unsigned long int.

6. Otherwise, if one of the operand is long int, the other will be converted to long int and the result will be long int.
7. Otherwise, if one of the operand is unsigned int, the other will be converted to unsigned int and the result will be unsigned int.
8. If none of the above conditions applies, then both the operands will be converted to int, and the result will be int.

2.13.2. Unary Operators

The class of operators that act upon a single operand to produce a new value are known as unary operators. The frequently used unary operators in C are (the others will be discussed later in the book):

- Unary Minus
- Increment and Decrement operator
- sizeof operator
- Cast operator

Unary Minus (-)

The operand of unary – operator must have arithmetic type, and the result is the negative of its operand. The operand can be numerical constant, variable, or expression. Note that the unary – operator is distinctly different from the arithmetic binary operator which denotes subtraction (-). The subtraction operator requires two separate operands.

Here are several examples which illustrate the use of the unary minus operation.

```
-123      -(x+y)      z = w+x* -y      flip = -net      -3*(x+y)
```

Increment (++) and Decrement (--) Operators

Increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1. The operand used with each of these operations must be a single variable.

The ++ and -- operators can each be utilized in two different ways, depending on whether the operator is written before or after the operand. If the operator precedes the operand (e.g., ++i), then the operand will be altered

in value before it is utilized for its intended purpose within the program. If, however, the operator follows the operand (e.g., i++), then the value of the operand will be altered after it is utilized.

As an example, suppose that n is an integer variable that has been assigned a value of 7. The statement

```
a = ++n;
```

first increments and then assign the value 8 to a. But the statement

```
a = n++;
```

first assigns the value 7 to a and then increments n to 8. In both cases, though, the end result is that n is assigned the value 8.

Now consider another example. Suppose that a C program includes an integer variable i whose initial value is 1. Suppose the program includes the following three printf statements.

```
printf("i=%d\n",i);
printf("i=%d\n",++i);
printf("i=%d\n",i);
```

These printf statements will generate the following three lines of output

```
i=1
i=2
i=2
```

The first statement causes the original value of i to be displayed. The second statement increments i and then displays its value. The final value of i is displayed by the last statement.

Now suppose that the program includes the following three printf statements, rather than the three statements given above.

```
printf("i=%d\n",i);
printf("i=%d\n",i++);
printf("i=%d\n",i);
```

The first and third statements are identical to those shown above. In the second statement, however, the unary operator follows the integer variable rather than precedes it. These statements will generate the following three lines of outputs.

```
i=1
i=1
i=2
```

The first statement causes the original value of i to be displayed, as before. The second statement causes the current value of I (1) to be displayed and then incremented to 2. The final value of i (2) is displayed by the last statement.

The sizeof Operator

The sizeof operator returns the size of its operand in bytes. The sizeof operator always precedes its operand. The operand may be an expression, or it may be a cast. This operator allows the determination of the number bytes allocated to various types of data items. This information can be very useful when transferring a program to a different computer or to a new version of C. It also used for dynamic memory allocation.

Suppose that i is an integer variable, x is a floating-point variable, d is a double precision variable and c is character-type variable. The statements

```
printf("Size of Integer        : %d",sizeof i);
printf("Size of Floating-point : %d",sizeof x);
printf("Size of Double         : %d",sizeof d);
printf("Size of Character      : %d",sizeof c);
```

might generate the following output

```
Size of Integer        : 2
Size of Floating-point : 4
Size of Double         : 8
Size of Character      : 1
```

Thus, this version of C allocates 2 bytes to each integer quantity, 4 bytes to each floating-point quantity, 8 bytes to each double-precision quantity, and 1 byte to each character.

Another way to generate the same information is to use a cast rather than a variable within each printf statement. Thus, the printf statement could have been written as

```
printf("Size of Integer        : %d",sizeof  (integer));
printf("Size of Floating-point : %d",sizeof (float));
printf("Size of Double         : %d",sizeof (double));
printf("Size of Character      : %d",sizeof (char));
```

These printf statements will generate the same output as that shown above.

The Cast Operator

Cast operator can be used to explicitly convert the value of an expression to a different data type. To do so, the name of the data type to which the conversion is to be made (such as int or float) is enclosed in parentheses and placed directly to the left of the expression whose value to be converted. The word 'cast' never is actually used. Its general format is

(data type) expression

Suppose that f is a floating-point variable and i is an integer variable. expression f % i is invalid, because the first operand is a floating-point constant rather than an integer. However, it could be written as (int)f % i. The data type associated with the expression itself is not changed by a cast. Rather, it is the value of the expression that undergoes type conversion wherever the cast appears.

Other examples are

(int) 3.1415 (int)n (float) (4*a)/b

C includes several other unary operators. They will be discussed in later units of this book.

2.13.3. Relational and Equality Operators

Relational and equality operators are symbols that are used to test the relationship between two quantities such as variables, constants or expressions.

There are four relational operators in C. They are

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Closely associated with the relational operators are the following two equality operators.

Operator	Meaning
==	Equal to
!=	Not equal to

These six relational operators are used to form logical expressions, which represent conditions that are either true or false. The resulting expression will be of type integer, since true is represented by integer value 1 and false represented by the value of 0.

Suppose that i, j and k are integer variables whose values are 1, 2, and 3, respectively. Several logical expressions involving these variables are shown below.

Expression	Interpretation	Value

i<j	true	1
(i+j)>=k	true	1
(j+k)>(i+5)	false	0
k!=3	false	0
j==1	false	0
j==2	true	1

When carrying out relational and equality operations, operands that differ in type will be converted in accordance with the rules discussed earlier.

Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5, and c is a character variable that represents the character ‘w’. Several logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
f>5	true	1
(i+f)<=10	false	0
c==119	true	1
c!=’p’	true	1
c>= 10*(i+f)	false	0

2.13.4. Logical Operators

Logical operators are the symbols that are used to combine or negate expressions containing relational operators. There are three logical operators in C

Operator	Meaning
&&	logical and
||	logical or
!	logical not

The logical operators act upon operands that are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex conditions that are either true or false. The result of the logical and operation will be true only if both operands are true, where as the result of a logical or operation will be false only if both operands are false. The unary operator logical not is used to negate the value of a logical expression, i.e., it causes an expression that is originally true to become false, and vice versa.

Suppose that i is an integer variable whose value is 7, f is floating-point variable whose value is 5.5 and c is character-type variable that represents the character 'w'. Several complex logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
(i>=6)&&(c=='w')	true	1
(i>=6)‖(c==119)	true	1
(f<11)&&(i>100)	false	0
c!='p')‖((i+f)<=10)	false	0
!(f>5)	false	0
!(i>(f+1))	false	0

2.13.5. Assignment Operators

An assignment operator is used to form an assignment expression, which assigns the value of an expression to an identifier. The most commonly used assignment operator is =. Assignment expressions that make use of this operator are written in the form

identifier = expression

where identifier generally represents a variable, and expression represents a constant, a variable or a more complex expression.

Here are some typical assignment expressions that make use of = operator.

a=3
x=y
delta=0.0001
area=length*breadth

Remember that the assignment operator = and the equality operator == are distinctly different. The assignment operator is used to assign a value to an identifier, where as the equality operator is used to determine if two expressions have the same value. Assignment expressions are often referred to as assignment statements, since they are usually written as complete statements. If two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of the identifier on the left. The entire assignment expression will then be of this data type. Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example,

- A floating-point value may be truncated if assigned to an integer identifier.
- A double precision may be rounded if assigned to floating-point identifier.
- An integer quantity may be altered if assigned to a short integer identifier or a character identifier.

In the following assignment expressions, suppose that i is an integer-type variable.

Expression	Value
i = 3.3	3
i = 3.9	3
i = -3.9	-3

Now suppose that i and j are both integer-type variables and j has been assigned a value of 5. Several assignment expressions that make use of these two variables are shown below.

Expression	Value
i = j	5
i = j/2	2
i = 2*j/2	5
i = 2*(j/2)	4

Finally assume that i is integer type variable, and the ASCII character set applies.

Expression	Value
i = 'x'	120
i = '0'	48
i = ('x'-'0')/3	24
i = ('y'-'0')/3	24

Multiple assignments of the form

identifier1 = identifier2 = ... = expression

are permissible in C. In such situations, the assignments are carried out from right to left. Thus the multiple assignments

identifier1 = identifier2 = expression

is equivalent to

identifier1 = (identifier2 = expression)

and so on, with right-to-left nesting for additional multiple assignments.
Suppose that i and j are integer variables. The multiple assignment expression

i = j = 5

will cause the integer value 5 to be assigned to both i and j. (To be more precise, 5 is first assigned to j, the value of j is then assigned to i)
Similarly the multiple assignment expression

i = j = 5.9

will cause the integer value 5 to be assigned to both i and j. Remember that the truncation occurs when the floating-point value 5.9 is assigned to i.

C contains the following five additional assignment operators

+= , −= , *= , /= , %=

To see how they are used, consider the first operator, +=. The assignment expression

expression1+=expression2

is equivalent to

expression1=expression1+expression2

similarly, the assignment expression

expression1−=expression2

is equivalent to

expression1=expression1−expression2

and so on for all five operators.
Usually expression1 is an identifier, such as variable or an array element.

Suppose that i and j are integer variables whose values are 5 and 7, and f and g are floating-point variables whose values are 5.5 and -3.25. Several assignment expressions that make use of these variables are shown below.

Expression	Equivalent expression	Final value
i += 5	i = i+5	10
f –= g	f = f-g	8.75
j *= (i-3)	j = j*(i-3)	14
f /= 3	f = f/3	1.8333
i % = (j-2)	i = i%(j-2)	0

2.13.6. The Conditional Operator

The conditional operator is represented by ? and : symbols. The conditional operator is a ternary operator, meaning it operates upon three values. Simple conditional operations can be carried out with conditional operator (?:). An expression that makes use of the conditional operator is called conditional expression. A conditional expression is written in the form

expression1 ? expression2 : expression3

The expression1 is interpreted as a Boolean expression. It is a sort of switch, in that it determines which of the two expressions, expression2 or expression3, is to be evaluated.

When evaluating a conditional expression, expression1 is evaluated first. If expression1 is true (i.e., if its value is non zero), then expression2 is evaluated and this becomes the value of the conditional expression. However, if expression1 is false (i.e., if its value is zero), the expression3 is evaluated and this becomes the value of the conditional expression. Note that

only one of the embedded expressions (either expression2 or expression3) is evaluated when determining the value of a conditional expression.

For example, in the conditional expression shown below, assume that i is an integer variable.

$$(i<0) \ ? \ 0 : 100$$

The expression (i<0) is evaluated first. If it is true, the entire conditional expression takes on the value 0. Otherwise, the entire conditional expression takes on the value 100.

In the following conditional expression, assume that f and g are floating-point variables.

$$(f < g) \ ? \ f : g$$

This conditional expression takes on the value of f if f is less than g; otherwise the conditional expression takes on the value of g. In other words, the conditional expression returns the value of the smaller of the two variables.

If the operands (i.e., expression2 and expression3) differ in type, then the resulting data type of the conditional expression will be determined by the rules given earlier. Now suppose that i is an integer variable, and f and g are floating-point variables. The conditional expression

$$(f < g) \ ? \ i : g$$

involves both integer and floating-point operands. Thus the resulting expression will be floating-point, even if the value of i is selected as the value of the expression.

Conditional expressions frequently appear on the right-hand side of a simple assignment statement. The resulting value of the conditional expression will be assigned to the identifier on the left.

Here is an assignment statement that contains an assignment expression on the right-hand side.

$$\text{min} \ = (f < g) \ ? \ f : g$$

this assignment expression causes the value of the smaller of f and g to be assigned to min.

2.14. Operator Precedence and Associativity

Operator precedence determines the order in which the operators are to be evaluated in a complex expression. The operators in C are grouped hierarchically according to their precedence. Operations with a higher precedence are carried out before operation having the lower precedence.

However the natural order of precedence can be altered through the use of parentheses.

The associativity determines the order in which consecutive operations within the same precedence group are carried out. The precedence and associativity of all operators in C are summarized below.

Precedence Group		Operators	Associativity
1.	Function, array, structure member, pointer to structure member	() [] . –>	L→R
2.	Unary operations	- ++ -- ! ~ * & sizeof (type)	R→L
3.	Arithmetic multiply, divide and modulus	* / %	L→R
4.	Arithmetic add and subtract	+ -	L→R
5.	Bitwise shift operators	<< >>	L→R
6.	Relational operators	< <= > >=	L→R
7.	Equality operators	== !=	L→R
8.	Bitwise AND	&	L→R
9.	Bitwise Exclusive OR	^	L→R
10.	Bitwise OR	|	L→R
11.	Logical AND	&&	L→R
12.	Logical OR	||	L→R
13.	Conditional operator	?:	R→L
14.	Assignment operators	= += –= *= /= %= &= ^= |= <<= >>=	R→L
15.	Comma operator	,	L→R

2.15. Library Functions

Technically speaking, one can create a useful, functional C program consist of solely of statements involving only C keywords. However, this is quite rare because C does not provide keywords that perform such things as input/output operations, high-level mathematical computations, or character handling. As a result, most programs include calls to various functions contained in C's standard library. All C compilers come with a standard library of functions that carry out various commonly used operations or calculations. Library functions that are functionally similar are usually grouped together as object programs in separate library files. These library files are supplied as a part of each C compiler. A library function is accessed by simply 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. Some commonly used library functions are

Function	Type	Purpose	Include File
abs(i)	int	Return the absolute value of i.	stdlib.h
ciel(d)	double	Return a value rounded up to the next higher integer.	math.h
cos(d)	double	Return the cosine of d	math.h
exp(d)	double	Raise e to the power d	math.h
fabs(d)	double	Return the absolute value of d	math.h
fclose(f)	int	Close file f. Return 0 if file is successfully closed	stdio.h
feof(f)	int	Determine if an end-of-file condition has been reached. If so, return a nonzero value; otherwise, return 0.	stdio.h
fgetc(f)	int	Enter a single character from a file.	stdio.h
fmod(d1,d2)	double	Return the remainder of d1/d2	math.h
floor(d)	double	Round down to the next integer value	math.h
fmod(d1,d2)	double	Return the remainder of d1/d2 with same sign as d1	math.h
fopen(s1,s2)	file*	Open file named s1 of type s2. Return a pointer to file.	stdio.h
fprintf(f,…)	int	Send data items to file f	stdio.h
fputc(c,f)	int	Send a single character to file f.	stdio.h
fputs(s,f)	int	Send a string s to file f.	stdio.h
fsacnf(f,…)	int	Read data items from file f.	stdio.h
getc(f)	int	Read a single character from file f.	stdio.h
getchar()	int	Enter a character from the keyboard	stdio.h
gets(s)	char*	Enter string s from the standard input device	stdio.h
isalnum(c)	int	Determine if argument is alphanumeric. Return a nonzero value if true; 0 otherwise	ctype.h
isalpha(c)	int	Determine if argument is alphabetic. Return a nonzero value if true; 0 otherwise	ctype.h
isascii(c)	int	Determine if argument is an ASCII character. Return a nonzero value if true; 0 otherwise	ctype.h
iscntrl(c)	int	Determine if argument is an ASCII control character. Return a nonzero value if true; 0 otherwise	ctype.h
isdigit(c)	int	Determine if argument is a decimal digit.	ctype.h

		Return a nonzero value if true; 0 otherwise	
islower(c)	int	Determine if argument is lowercase. Return a nonzero value if true; 0 otherwise	ctype.h
isodigit(c)	int	Determine if argument is an octal digit. Return a nonzero value if true; 0 otherwise	ctype.h
isprint(c)	int	Determine if argument is a printing ASCII character. Return a nonzero value if true; 0 otherwise	ctype.h
ispunct(c)	int	Determine if argument is a punctuation character. Return a nonzero value if true; 0 otherwise	ctype.h
isspace(c)	int	Determine if argument is a white space character. Return a nonzero value if true; 0 otherwise	ctype.h
isupper(c)	int	Determine if argument is uppercase. Return a nonzero value if true; 0 otherwise	ctype.h
isxdigit(c)	int	Determine if argument is a hexadecimal digit. Return a nonzero value if true; 0 otherwise	ctype.h
labs(l)	long int	Return the absolute value of l	math.h
log(d)	double	Return the natural logarithm of d	math.h
log10(d)	double	Return the logarithm (base 10) of d	math.h
malloc(u)	void*	Allocate U bytes of memory. Return a pointer to beginning of the allocated space	stdlib.h
pow(d1,d2)	double	Return d1 raised to the power d2	math.h
printf(...)	int	Send the data item to the monitor	stdio.h
putc(c,f)	int	Send a single character to file.	stdio.h
putchar(c)	char	Send a character to the monitor	stdio.h
puts(s)	int	Send string s to the standard output device	stdio.h
rand()	int	Return a random positive integer	stdlib.h
rewind(f)	void	Move pointer to the beginning of the file	stdio.h
scanf(...)	int	Enter the data item from the keyboard	stdio.h
sin(d)	double	Return the sine of d	math.h
sqrt(d)	double	Return the square root of d	math.h
strcmp(s1,s2)	int	Compare two strings lexicographically. Return a negative value if s1<s2; 0 if s1 and s2 are identical; and a positive value if s1>s2	string.h
strcmpi(s1,s2)	int	Compare two strings lexicographically, without regard to case. Return a negative value if s1<s2; 0 if s1 and s2 are identical; and a positive value if s1>s2	string.h
strcpy(s1,s2)	char*	Copy string s2 to string s1	string.h
strlen(s)	int	Return the number of character in a string	string.h
tan(d)	double	Return the tangent of d	math.h

toascii(c)	char	Convert c to ASCII	ctype.h
tolower(c)	char	Convert c to lower-case	ctype.h or stdlib.h
toupper(c)	char	Convert c to upper-case	ctype.h or stdlib.h

Note: Type refers to the data type of the quantity that is returned by the function.

* denotes a pointer	c denotes a character type argument
i denotes an integer argument	d denotes a double-precision argument
f denotes a file argument	l denote long int argument
s denotes a string argument	u denote a unsigned int argument

In order to use a library function it may be necessary to include certain files that contain information about the library functions. Such information is normally kept in header files, such as stdio.h, stdlib.h, math.h, string.h, etc. The header files can be accessed by the pre-processor statement #include.

#include<filename>

where the filename represents the name of the header file.

2.16. Use of Comments

Comments should always be included within a C program. If written properly, comments can provide a useful overview of the general programming logic. They also delineate major segments of a program, identify certain key items within the program and provide other useful information about the program. Generally, the comments need not be extensive; a few well-placed comments can shed a great deal of light on an otherwise obscure program. Such comments can be of great use to the original programmer as well as to other persons trying to read and understand a program, since most programmers do not remember the details of their own programs over a period of time. This is especially true of programs that are long and complicated.

A comment is normally placed within the delimiters /* and */. A comment can appear anywhere within a program, and they may be of any length. A comment may start on the same line as another statement or on a line of its own. It may also extend over any number of lines if the terminating delimiter is placed at the appropriate point.

Questions

3 Mark Questions

1. Explain the structure of a C program with an example.
2. Which characters comprise the C character set?
3. What is an identifier? Summarize the rules for naming identifiers.
4. What is a Keyword? What restriction applies to their use?
5. Name and describe the basic data types in C.
6. What is meant by data type qualifier? Name and describe the four data type qualifiers.
7. Name and describe the four data type qualifiers. To which data type can each qualifier be applied?
8. Name and describe the four types of constants in C.
9. What is an integer constant? Name and describe the different types of integer constants in C.
10. How do the unsigned and long integer constants differ from ordinary integer constants? How can they be written and identified.
11. Describe two different ways that floating-point constants can be written.
12. How can single-precision and long floating-point constants can be written and identified in C?
13. What is a character constant? Do character constant represent numerical values?
14. What is an escape sequence? What is its purpose?
15. What is a string constant? How do string constants differ from character constants?
16. Differentiate between the constants 'A' and "A" in C.
17. What is a variable? Summarize the rules for naming a variable.
18. What is the purpose of type declaration? What does a type declaration consist of?
19. How are initial values assigned to variable within a type declaration?
20. What is an expression? What are its components?
21. What are the different classes of statements in C? Explain.
22. List different types operators used in C.
23. What is meant by operator precedence? Summarize the precedence of C operators.
24. What is meant by associativity of operators? Summarize the associativity of C operators.
25. What are unary operators? List and explain unary operators used in C.
26. What are unary operators? What is their precedence and associativity?
27. List and explain the arithmetic operators in C. What are the relative priorities of arithmetic operators?
28. Explain two different ways to utilize increment and decrement operators. How do the two methods differ?

29. Differentiate between the expressions a++ and ++a in C. Give an example of their use.
30. Differentiate between the expressions a-- and --a in C. Give an example of their use.
31. Explain the function of the cast operator. Give examples of its use.
32. What is the function of sizeof operator? Give examples of its use?
33. What are relational and equality operators? List and explain the relational and equality operators in C.
34. What are logical operators? List and explain the logical operators in C.
35. Explain the logical NOT operator in C. What is its purpose?
36. What is the relative precedence of relational, equality and logical operators with respect to one another?
37. Describe the different forms of assignment operator.
38. How can multiple assignments be written in C? In what order will the assignments be carried out?
39. What is the precedence of assignment operators relative to other operators? What is their associativity?
40. What is conditional operator? Explain its use.
41. What are library functions? Explain their purpose. Give examples.
42. How can comments be included in C programs? Explain the advantages of using comments in a program.

8 Mark Question

1. Describe various classes of operators in C

Problems

1. Determine which of the following are valid identifiers. If invalid explain why.

(a)	record1	(f)	name
(b)	1record	(g)	name and address
(c)	file_3	(h)	name_and_address
(d)	return	(i)	name-and-address
(e)	$tax	(j)	123-45-6789

2. Determine which of the following numerical values are valid constants. If a constant is valid, specify whether it is integer or real. Also, specify the base for each valid integer constant.

(a)	.5	(e)	12345678	(i)	0515

(b)	27,822	(f)	12345678L	(j)	0188CDF
(c)	9.3e12	(g)	0.8E+0.8	(k)	0XBCFDAL
(d)	9.3e-12	(h)	0.8E 8	(l)	0x8783ha

3. Determine which of the following are valid character constants.

(a)	'a'	(f)	'\a'
(b)	'$'	(g)	'T'
(c)	'\n'	(h)	'\0'
(d)	'/n'	(i)	'xyz'
(e)	'\\'	(j)	'\052'

4. Determine which of the following are valid string constants.

(a) '8.15 P.M.'
(b) "Red, White and Blue"
(c) "Name
(d) "Chapt. 3 (con\'d)"
(e) "New Delhi"
(f) "1.3e-12"
(g) "Teacher said, "Please don't sleep in the class""

5. Write appropriate declarations for each group of variables and arrays

(a) Integer variables: p, q
Floating-point variables: x, y, z
Character variables: a, b, c

(b) Floating-point variables: root1, root2
Long integer variable: counter
Short integer variable: flag

(c) Integer variable: index
Unsigned integer variable: cust_no
Double-precision variables: gross, tax, net

(d) Character variables: current, last
Unsigned integer variable: count
Floating-point variable: error

(d) Character variables: first, last
80-element character array: message

6. Write appropriate declarations and assign the given initial values for each group of variables and arrays.

(a) Floating-point variables: a = 8.2, b = 0.005
Integer variables: x = 129, y = 87, z = -22
Character variables: c1 = 'w', c2 = '&'

(b) Double-precision variable: d1 = 2.88×10^{-8}, d2 = -8.4×10^{5}
Integer variables: u = 711 (octal), v = ffff (hexadecimal)

(c) Long integer variable: big = 123456789
Double-precision variable: c = 0.3333333333
Character variables: eol = newline character

7. Explain the purpose of each of the following expressions.
   (a) a-b
   (b) a*(b+c)
   (c) d=a*(b+c)
   (d) a>=b
   (e) (a%5)==0
   (f) a<(b/c)
   (g) –a

8. Suppose a,b, and c are integer variables that have assigned the values a = 8, b = 3 and c = -5. Determine the value each of the following arithmetic expressions.

(a)	a+b+c	(f)	a%c
(b)	2*b+3*(a-c)	(g)	a*b/c
(c)	a/b	(h)	a*(b/c)
(d)	a%b	(i)	(a*c)%b
(e)	a/c	(j)	a*(c%b)

9. Suppose x, y and z are floating-point variables that have been assigned the values x = 8.8, y = 3.5 and z = -5.2. Determine the value of each of the following arithmetic expressions.

(a)	x+y+z	(e)	x/(y+z)
(b)	2*y+3*(x-z)	(f)	(x/y)+z
(c)	x/y	(g)	2*x/3*y
(d)	x%y	(h)	2*x/(3*y)

10. Suppose c1, c2 and c3 are character-type variables that have been assigned the characters E, 5 and ?, respectively. Determine the numerical value of the following expressions, based upon the ASCII character set.

(a)	c1	(f)	c1%c3
(b)	c1-c2+c3	(g)	‘2’+‘2’
(c)	c2-2	(h)	(c1/c2)*c3
(d)	c2-‘2’	(i)	3*c2
(e)	c3+‘#’	(j)	‘3’*c2

11. A C program contains the following declarations:

```
int i,j;
ling ix;
short s;
float x;
double dx;
char c;
```

Determine the data type of each of the following expressions.

(a)	i+c	(f)	s+j

(b) x+c
(c) dx+x
(d) ((int)dx)+ix
(g) ix+j
(h) s+c
(i) ix+c
(j) i+x

12. A C program contains the following declarations and initial assignments:

```
int i = 8, j = 5;
float x = 0.005, y = -0.01;
char c = 'c', d = 'd';
```

Determine the value of each of the following expressions.

(a) (3*i-2*j)%(2*d-c)
(b) 2*((i/5)+(4*(j-3))%(i+j-2)
(c) (i-3*j)%(c+2*d)/(x-y)
(d) –(i+j)
(e) ++i
(f) i++
(g) −−j
(h) ++x
(i) y--
(j) i<=j
(k) c>d
(l) x>=0
(m) x<y
(n) j!=6
(o) c==99
(p) 5*(i+j)>'c'
(q) (2*x+y)==0
(r) 2*x+(y==0)
(s) 2*x+y==0
(t) !(i<=j)
(u) !(c==99)
(v) !(x>0)
(w) (i>0)&&(j<5)
(x) (i>0)||(j<5)
(y) (x>y)&&(i>0)||(j<5)
(z) (x>y) &&(i>0)&&(j<5)

13. A C program contains the following declarations and initial assignments

```
int i = 8, j = 5, k;
float x = 0.005, y = -0.01, z;
char a,b,c = 'c', d = 'd';
```

Determine the value of each of the following assignment expressions. Use the values originally assigned to the variables for each expression.

(a)	k=(i+j)	(l)	y−=x
(b)	z=(x+y)	(m)	x *=2
(c)	i=j	(n)	i/=j
(d)	k=(x+y)	(o)	i%=j
(e)	k=c	(p)	i+=(j-2)
(f)	z=i/j	(q)	k=(j==5)?i:j
(g)	a=b=d	(r)	k=(j>5)?i:j
(h)	i=j=1.1	(s)	z=(y>=0)?y:0
(i)	z=k=x	(t)	z=(y>=0)?y:0
(j)	k=z=x	(u)	a=(c<d)?c:d
(k)	i+=2	(v)	i−=(j>0)?j:0

Answers

1.
   (a) Valid
   (b) An identifier must begin with a letter
   (c) Valid
   (d) return is a reserved word
   (e) An identifier must begin with a letter
   (f) Valid
   (g) Blank spaces are not allowed
   (h) Valid
   (i) Minus sign is not allowed
   (j) An identifier must begin with a letter or an underscore

2.
   (a) Valid (real)
   (b) Illegal character (,)
   (c) Valid (real)
   (d) Valid (real)
   (e) Valid (decimal integer)
   (f) Valid (long integer)
   (g) Valid (real)
   (h) Illegal character (blank space)
   (i) Valid (octal constant)
   (j) Illegal characters (C,D,F), if intended an octal constant. If intended as an octal constant an x must be included (i.e., 0x18CDF)
   (k) Valid (hexadecimal long integer)
   (l) Illegal character (h)

3.
   (a) Valid
   (b) Valid
   (c) Valid
   (d) Escape sequences must be written with a backslash
   (e) Valid
   (f) Valid
   (g) Valid
   (h) Valid (Null character escape sequence)
   (i) A character constant cannot consist of multiple characters
   (j) Valid (Octal escape sequence). Note that octal 52 is equivalent to decimal 42. In the ASCII character set, this value represents an asterisk (*).

4.
   (a) A string constant must be enclosed in double quotation mark
   (b) Valid
   (c) Trailing quotation mark is missing.
   (d) Valid

(e) Valid
(f) Valid
(g) Quotation marks and (optionally) the apostrophe within the string must be expressed as escape sequence, i.e., “Teacher said, \” Please don\’t sleep in the class\””

5.

(a)
```
int p,q;
float x,y,z;
char a,,b,c;
```

(b)
```
float root1,root2;
long counter;
short flag;
```

(c)
```
int index;
unsigned cust_no;
double gross, tax, net;
```

(d)
```
char current,last;
unsigned count;
float error;
```

(e)
```
char first,last;
char message[80];
```

6.

(a)
```
float a = -8.2, b = 0.005;
int x = 129, y = 87, z = -22;
char c1 = ‘w’ , c2 = ‘&’;
```

(b)
```
double d1 = 2.88e-8, d2 = -8.4e5;
int u = 0711, v = 0xffff;
short flag;
```

(c)
```
long big = 123456789L;
double c = 0.3333333333;
char eol = ‘\n’;
```

7.

(b) Subtract the value of b from the value of a
(c) Add the values of b and c, then multiply the sum by the value of a.
(d) Add the values of b and c then multiply the sum by the value of a Then assign the result to d.
(e) Determine whether or not the value of a is greater than or equal to the value of b. The result will be either true or false, represented by the value 1 (true) or 0 (false)
(f) Divide the value of a by the value of c, and determine whether or not the remainder is equal to zero. The result will be either true or false.
(g) Divide the value of b by the value of c, and determine whether or not the value of a is less than the quotient. The result will be either true or false.
(h) Decrement the value of a; i.e., decrease the value of a by 1.

8.

(a) 6	(b) 45	(c) 2	(d) 2	(e) -1
(f) 3	(g) -4	(h) 0	(i) -1	(j) -16

9.

(a) 7.1	(b) 49	(c) 2.51429	(d) Invalid	(e) –517647
(f) –2.68571	(g) 20.53333	(h) 1.67619		

10.

(a) 69	(b) 79	(c) 53	(d) 3	(e) 98
(f) 6	(g) 100	(h) 63	(i) 159	(j) 2703

11.

(a) integer	(b) float	(c) double precision
(d) long integer	(e) float (or double)	(f) integer
(g) long integer	(h) integer	(i) long integer

12.

(a) 14	(b) 18	(c) –466.667	(d) -13	(e) 9
(f) 9	(g) 4	(h) 1.005	(i) –1.01	(j) –0
(k) 0	(l) 1	(m) 0	(n) 1	(o) 1
(p) 0	(q) 1	(r) 0.01	(s) 1	(t) 1
(u) 0	(v) 0	(w) 0	(x) 1	(y) 1
(z) 0				

13.

(a) k=13	(b) z=-0.005	(c) i=5	(d) k=0
(e) k=99	(f) z=1.0	(g) b=100,a=100	(h) j=1,i=1
(i) k=0,z=0.0	(j) z=0.005,k=0	(k) i=10	(l) y=-0.015
(m) x=0.010	(n) i=1	(o) i=3	(p) i=11
(q) k=8	(r) k=5	(s) z=0.005	(t) z=0.0
(u) a='c'	(v) i=3		

UNIT III

DATA INPUT & OUTPUT FUNCTIONS

3.1. Introduction

One of the primary advantages of the modern computers is their ability to communicate with the user during program execution. This feature enables the programmer to enter the values into variables as the occasion demands, without having to change the program itself. The advantage of this option is that execution stops each time the program needs to have a new value entered. Once the required value has been entered and the return key is pressed, execution resumes form the point at which it stopped. Thus a program written in C may requires data to be read into variables and data stored in variables need to be displayed. As a programming language, C does not provide any keyword or statement that perform input/output. Instead, input and output are accomplished through library functions provided by the C compilers. C's input/output system is quite large and consists of several different functions. The header for I/O functions is <stdio.h>. There are both console and file I/O functions. This chapter examines the console I/O functions that read input from the keyboard and provide output to the screen. The file I/O system will be discussed later in Unit XIII.

3.2. Single Character Input - getchar(), getch() and getche() Functions

Single characters can be entered into the computer using the C library function getchar(). The get getchar() function waits until a key is pressed and then returns its value. The key press is also echoed to the screen. The prototype of getchar() is

```
int getchar(void);
```

As its prototype shows, the getchar() function is declared as returning an integer even though a character is read. However, one can assign this value to a char variable, as is usually done, because the character is contained in the low-order byte (the high-order byte is usually zero). In general terms, a function reference would be written as

```
character variable = getchar();
```

where character variable refers to some previously declared character variable.

For example, a C program contains the following statements.

```
char c;
....
```

```
c = getchar();
```

The first statement declares that c is character type variable. The second statement causes a single character to be entered from the keyboard and then assigned to c.

The getchar() returns an EOF if an error occurs. The symbolic constant EOF is defined in <stdio.h> and is often equal to –1.

The getchar() function can also be used to read multicharacter string, by reading one character at a time within a multipass loop.

There are some potential problems with getchar() function. For many compilers, getchar() is implemented in such a way that it buffers input until Enter Key is pressed. This is called a line buffered input. One has to press Enter before any character is returned. Also, since getchar() inputs only one character each time it is called, the line buffering may leave one or more characters in waiting in the input queue, which is annoying in interactive environments. Even though it is permissible for getchar() to be implemented as an interactive function, it seldom is. For example, the following program is supposed to input characters from the keyboard and display it in reverse case. The program should terminate only when a period has entered.

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
main()
{
    char c;
    do
    {
        c=getchar();
        if(islower(c))
            c=toupper(c);
        else
            c=tolower(c);
        putchar(c);
    }
    while(c!='.');
}
```

This program may not behave as you expected because of the reasons discussed above.

Two of the most common alternatives to getchar() function are getch() and getche(). They have the following prototypes

```
int getch(void);
int getche(void);
```

For most compilers, the prototypes for these functions are found in the console header file <conio.h>

The getchar() function waits for a key press after which it returns immediately. It does not echo each character to the screen. The getche() is same as getch(), but the key is echoed. These two functions are commonly used instead of getchar() when a character needs to read from the keyboard in an interactive program. For example, the previous program is shown below using getche() instead of getchar().

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
main()
{     char c;
      do
      {       c=getche();
       if(islower(c))
              c=toupper(c);
       else
              c=tolower(c);
       putchar(c);
      }
      while(c!='.');
}
```

When you run this version of the program, each time you press a key, it is echoed on the screen and also immediately transmitted to the program and displayed in reverse case. Input is no longer line buffered.

3.3. The putchar(c) Function

Single characters can be displayed on the screen using C library function putchar(c). The prototype for putchar() function is

int putchar(int c);

In putchar function, even though it is declared as taking an integer type parameter, one can call it using a character assignment. Only the low-order byte of its parameter is actually output to the screen (Remember that an integer has two bytes of memory associated with it and the character is contained in the low-order byte). The putchar function returns the character written or EOF if an error occurs.

The putchar() function transmits a single character to the monitor. The character being transmitted will normally be represented as a character type variable. In general, a function reference would be written as

putchar(character variable)

where character variable refers to some previously declared character variable.

For example, a C program contains the following statements.

```
char c;
.....
putchar(c);
```

The first statement declares that c is a character type variable. The second statement causes the current value of c to be displayed on the screen.

The putchar function can be used to output multiple character strings, by displaying one character at a time within a multipass loop.

3.4. The scanf(...) Function

The scanf 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.

In general terms, the scanf function is written as

scanf(format string, arg1,arg2,...,argn)

where the format string refer to a string containing some required formatting information, and arg1, arg2,..., argn are arguments that represent the individual data items. The arguments represent pointers that indicate the address of the data items within the computers memory.

The format string consists of individual group of characters, with one character group for each input data item. Each character group must begin with a percent (%) sign. In its simplest form, a single character group will consist of the % sign followed by a conversion character, which indicates the type of the corresponding data item.

Within the format string, multiple character groups can be contiguous, or they can be separated by white space characters (i.e., blank spaces, tab, or newline character). If white space characters are used to separate multiple character groups in the format string, all the consecutive white space characters in the data will be read but ignored.

The more frequently used conversion characters for data input are

Conversion Character	Meaning
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating-point value
f	Data item is a floating-point value
g	Data item is a floating-point value
h	Data item is a short integer
i	Data item is a decimal, octal, or hexadecimal integer
o	Data item is an octal integer
s	Data item is a string followed by a null character
u	Data item is an unsigned decimal integer
x	Data item is a hexadecimal integer
[...]	Data item is a string, which may include white space characters (scan list)

The arguments are written as variables or arrays, whose types match the corresponding character groups in the format string. The arguments are actually pointers that indicate where the data items are stored in computer's memory. Thus each variable name must be preceded by the address operator & (ampersand). However, array names should not begin with an ampersand, since the array name is really pointer to the first element in the array.
An example of scanf function is given below.

```
#include <stdio.h>
main()
{    char item_name[15];
     int item_code;
     float unit_price;
     ..........
     scanf("%s  %d  %f", item_name, &item_code, &unit_price);
     ..........
}
```

Within the scanf function, the format string is "%s %d %f". It contains three character groups. The first character group, %s, indicates that the first argument (item_name) represents a string. The second character group, %d, indicates that the second argument (&item_code) represents a decimal integer value, and the third character group, %f, indicates that the third argument (&unit_price) represents a floating-point value.
The above scanf function could have been written as

```
scanf("%s%d%f", item_name, &item_code, &unit_price);
```

with no white space characters. This is also valid, though the input data could be interpreted differently when using c-type conversion.

The actual data items must correspond to the arguments in the scanf function in number, in type and in order. Numeric data items are written in the same form as numeric constants, though octal values need not be preceded by a 0, and hexadecimal values need not be preceded by 0x or 0X. Floating-point values must include either a decimal point or an exponent or both.

If two or more data items are entered, they must be separated by white space characters, except when reading single characters. In such cases, a white space character within the input data will be interpreted as a data item. The data items may continue onto two or more lines, since the newline character is considered to be a white space character and can therefore separate consecutive data items. If the format string begins by reading a character type data item, it is generally good idea to precede the first conversion character with a blank space. This causes the scanf function to ignore any extraneous characters that may have been entered earlier, for example, by pressing the Enter key after entering a previous line of data.
For example, consider the following skeletal outline of a C program

```
#include <stdio.h>
main()
{  char item_name[15];
   int item_code;
   float unit_price;
   .....
   scanf(“  %s  %d  %f”, item_name, &item_code, &unit_price);
   .....
}
```

Notice the blank space that precedes %s. The following data items could be entered from the keyboard when the program is executed.

```
Computer  1234  32800.00
```

Thus, the characters that make up the string Computer would be assigned to the first eight elements of the array item_name, the integer value 1234 would be assigned to the item_code, and the floating-point value 32800.00 would be assigned to unit_price. The data items could also be entered in any of the following ways:

```
Computer        Computer                Computer  1234
1234            1234  32800.00          32800.00
32800.00
```

Now consider the following skeletal outline of a C program.

```
#include<stdio.h>
main()
{  char c1,c2,c3;
   .............
   scanf(“  %c%c%c”,&c1, &c2, &c3);
   .............
```

```
}
```

If the input data consisted of

```
a b c
```

(with one blank space between the letters), then the following assignment will result.

```
c1=a,   c2=<blank space>,   c3=b
```

This problem can be avoided by writing the scanf function as

```
scanf(" %c %c %c",&c1, &c2, &c3);
```

with blank spaces separating the %c terms. Now the data items could be entered in any of the following ways:

(i) a
b
c

(ii) abc

(iii) a b c

(iv) a
b c

(v) a b
c

(vi) a
b
c

All these result in the following assignments

```
c1=a, c2=b, c3=c
```

The s-type conversion character applies to a string that is terminated by a white space character. Therefore, a string that includes white space characters cannot be entered in this manner. To work with strings that includes white space characters general-purpose format specifier, called scan set, can be employed. A scan set defines a set of characters. The sequence of characters is enclosed in square brackets, designated as [...]. White space characters may be included within the brackets, thus accommodating strings that contain such characters.

When the program is executed, successive characters will continue to be read from the keyboard as long as each input character matches one of the characters enclosed within the brackets. The order of the characters within the square brackets need not correspond to the order of the characters being entered. Input characters may be repeated. The string will terminate, however, once an input character is encountered that does not match any of the characters within the brackets. A null character (\0) will then automatically be added to the end of the string.

The following example illustrates the scanf function to enter a string consisting of uppercase letters and blank spaces. The string will be of undetermined length, but it will be limited to 80 characters, including the null character that is added at the end.

```
#include <stdio.h>
main()
{  char text[80];
   .....
   scanf("  %[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", text);
   .....
}
```

If the string

THIS IS A SAMPLE TEXT

is entered from the keyboard when the program is executed, the entire string will be assigned to the array text, since the string is comprised entirely of uppercase letters and blank spaces. If the string were written as

This Is A Sample Text

however, then only the single character T would be assigned to text, since the first lowercase letter would interpreted as the first character beyond the string. It would, of course, be possible to include both uppercase and lowercase characters within the brackets, but this become cumbersome.

A variation of this feature is which is often more useful is to precede the characters within the square brackets by a circumflex (i.e., ^). This causes the subsequent characters within the brackets to be interpreted in the opposite manner. Thus, when the program is executed, successive characters will continue to be read from the keyboard as long as each input character does not match one of the characters enclosed within the brackets.

For example, if the characters within the brackets are simply the circumflex followed by a newline character, the string entered from the keyboard can contain any ASCII characters except the newline character. This is illustrated in following skeleton structure of a C program.

```
#include <stdio.h>
main()
{  char text[80];
   .....
   scanf("  %[ ^\n]", text);
   .....
}
```

Notice the blank space preceding %[^\n], to ignore any unwanted characters that may have been entered previously.

The consecutive non-whitespace characters that define a data item collectively define a field. It is possible to limit the number of characters in a data item by specifying a maximum field width for that data item. The maximum field width for a data item can be specified by placing an unsigned integer indicating the field width between the percent sign (%) and the conversion character. The data item may contain fewer characters than the specified field width. Any characters that extend beyond the specified field width will not be read. Such leftover characters may be incorrectly interpreted as the component of the next data item.

For example, consider the following skeletal outline of a C program.

```
#include<stdio.h>
main()
{  int a,b,c;
   .....
   scanf("  %3d  %3d  %3d",&a, &b, &c);
    .....
```

```
}
```

When the program is executed, three integer quantities will be entered from the keyboard. Suppose the input data items are entered as

```
1 2 3
```

Then the following assignment will result:

```
a=1,    b=2,    c=3
```

If the data had been entered as

```
123  456  789
```

Then the assignment would be

```
a=123,  b=456,  c=789
```

Now suppose that the data had been entered as

```
123456789
```

Then the assignment would be

a=123, b=456, c=789

as before, since the first three digits would be assigned to a, the next three digits to b, and the last three digits to c.

However, if the data had been entered as

```
1234  56  789
```

The resulting assignment would be

```
a=123,  b=4,    c=56
```

The remaining three digits (7,8 and 9) would be ignored, unless they were read by subsequent scanf function.

For example consider the following skeletal outline of a C program

```
#include<stdio.h>
main()
{  int a;
   float b;
   char c;
   ..............
   scanf(“  %3d  %6f   %c”,&a, &b, &c);
   ..............
}
```

If the data items are entered as

12 3456.789 a

when the program is executed, then 12 will be assigned to a, 3456.7 will be assigned to b and the character 8 will be assigned to c. The remaining two input characters (9 and a) will be ignored.

Certain conversion characters within the format string can be preceded by a single-letter prefix, which indicates the length of the

corresponding argument. For example, an l (lowercase L) is used to indicate either a signed or unsigned long integer argument, or a double precision argument. Similarly, an h is used to indicate a signed or unsigned short integer.

For example, consider the following skeletal outline of a C program

```
#include<stdio.h>
main()
{  short sa,sb;
   long la,lb;
   double da,db;
   ..............
   scanf(“  %hd  %ld  %lf”, &sa, &la, &da);
   scanf(“  %3ho  %7lx  %15le”, &sb, &lb, &db);
   ..............
}
```

The format string in the first scanf function indicates that the first data item will be assigned to a short decimal integer variable, the second will be assigned to a long decimal integer variable, and the third will be assigned to a double precision (long float) variable. The format string in the second scanf function indicates that the first data item will have a maximum field width of 3 characters and it will be assigned to a short octal integer variable, the second data item will have a maximum field width of 7 characters and it will be assigned to a long hexadecimal integer variable and the third data item will have a maximum field width of 15 characters and it will be assigned to a double precision variable.

It is possible to skip over a data item, without assigning it to the designated variable, by placing an asterisk (*) that follows the % sign within the appropriate format group. This feature is called assignment suppression. Assignment suppression is especially useful when one needs to process only a part of what is being entered. For example, given

```
scanf(“  %d  %*c   %d”, &a, &b);
```

One could enter the data 10,10. The comma would be correctly read, but not assigned to anything.

3.5. The printf(...) Function

Output data can be written from the computer onto a standard output device using the library function printf. This function can be used to output any combination of numerical values, single characters and strings. It is similar to the input function scanf, except that its purpose is to display data rather than to enter it into the computer. That is, the printf function moves data from the computer’s memory to the standard output device, where as

the scanf function enters data from the standard input device and store it in the computer's memory.
In general terms, the printf function is written as

printf(format string, arg1,arg2,...,argn)

where the format string refer to a string containing some required formatting information, and arg1, arg2,..., argn are arguments that represent the individual data items. The arguments can be written as constants, single variable, or array names, or more complex expressions. Function references may also be included. Unlike the scanf function, the arguments in the printf function do not represent memory addresses and therefore are not preceded by ampersands.

The format string consists of individual group of characters, with one character group for each output data item. Each character group must begin with a percent (%)sign. In its simplest form, an individual character group will consist of the % sign followed by a conversion character, which indicates the type of the corresponding data item.

Within the format string, multiple character groups can be contiguous, or they can be separated by other characters, including white space characters. These "other" characters are simply transferred directly to the output device, where they are displayed. Some of the more frequently used conversion characters for data input are

Conversion Character	Meaning
c	Data item is displayed as a single character
d	Data item is displayed as a signed decimal integer
e	Data item is displayed as a floating-point value with an exponent.
f	Data item is displayed as a floating-point value without an exponent.
g	Data item is displayed as a floating-point value using either e-type or f-type conversion, depending on the value. Trailing zeroes and trailing decimal points are not displayed.
i	Data item is displayed as a single decimal integer
o	Data item is displayed as an octal integer, without a leading zero
s	Data item is displayed as a string.
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer, without leading 0x

For example, the following C program illustrates the use of printf statement.

```
# include <math.h>
#include <stdio.h>
main()
{    float i=2.0,j=3.0;
     scanf("%f  %f  %f  %f  %f ", i, j, i+j, sqrt(i), sqrt(i+j);
}
```

The first two arguments within the printf function are single variables, the third argument is an arithmetic expression, and the last two arguments are function references.

The following skeletal outline of a C program illustrates how several different types of data can be displayed using the printf function.

```
#include <stdio.h>
main()
{     char item_name[15];
      int item_code;
      float unit_price;
      .....
      printf("%s  %d  %f", item_name,  item_code,  unit_price);
      .....
}
```

Within the printf function, the format string is "%s %d %f". It contains three character groups. The first character group, %s, indicates that the first argument (item_name) represents a string. The second character group, %d, indicates that the second argument (&item_code) represents a decimal integer value, and the third character group, %f, indicates that the third argument (&unit_price) represents a floating-point value. Note that the arguments are not preceded by ampersand

Now suppose that item_name, item_code and unit_price have been assigned the values Computer, 1234 and 32800.00, respectively, within the program. When the printf function is executed, the following output will be generated.

Computer 1234 32800.000000

The single space between data items is generated by the blank spaces that appear within the format string in the printf function.
Suppose the printf function had written as

printf("%s%d%f", item_name, item_code, unit_price);

This printf statement is syntactically valid, though it causes the output items to run together, as shown below.

Computer123432800.000000

The f-type conversion and e-type conversion are both used to output floating-point values. However, the latter causes an exponent to be included in the output, where as the former does not.

For example, the following C program generates the same floating-point output in different forms.

```
#include <stdio.h>
main()
{
   double x = 1234.56789, y = 0.123456789;
   printf("%f  %f  %f  %f\n", x, y, x*y, x/y);
   printf("%e  %e  %e  %e", x, y, x*y, x/y);
}
```

Both printf functions have the same arguments. However, the first printf function makes use of f-type conversion, whereas the second printf function uses e-type conversion. Also, notice the newline character (\n) in the first printf function. This causes the output to be displayed on separate lines.

When the program is executed, the following output is generated.

```
1234.567890 0.123457 152.415788 10000.000000
1.23457e+03 1.23457e-01 1.52416e+02 1.00000e+04
```

The first line of output shows the quantities represented by x, y, x*y and x/y in standard floating-point format, without exponents. The second line of output shows these same quantities in exponential format.

The printf function interprets the s-type conversion differently than the scanf function. In the printf function, s-type conversion is used to output a string that is terminated by a null character. Whitespace characters may be included within the string.

For example, the following C program will read in a line of text and then write it back out, just as it was entered. This program illustrates the syntactic differences in reading and writing a string that contains a variety of characters, including whitespace characters.

```
#include <stdio.h>
main()
{     char text[80];
      clrscr();
      scanf("  %[^\n]", text);
      printf("%s", text);
}
```

Notice the difference in the format strings within scanf function and printf function. Now suppose that the following string is entered from the keyboard when the program is executed.

Hello! Welcome to This Sample Program.

This string contains lowercase characters, uppercase characters, whitespace characters and some other special characters. The entire string can be entered with the single scanf function, as long as it is terminated by a newline character (by pressing the Enter key). The printf function will then

cause the entire string to be displayed on the standard output device, just as it had been entered.

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 width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters in the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width specification in the scanf function, which specifies a maximum field width.

The following C program illustrates the use of minimum field width feature.

```
#include<stdio.h>
main()
{ int x = 12345;
  float y=123.456789;
  clrscr();
  printf("%3d  %5d  %8d\n", x, x, x);
  printf("%3f  %10f  %13f", y, y, y);
  getch();
}
```

When the program is executed, the following output will be generated.

```
12345 12345    12345
123.456787 123.456787    123.456787
```

Besides the minimum field width specification, it is also possible to specify the maximum number of decimal places for a floating-point value, or the maximum number of characters for a string . This specification is known as precision. The precision is an unsigned integer that is always preceded by a decimal point. If a minimum field width is specified in addition to the precision, then the precision specification follows the field width specification. Both of these integer specifications precede the conversion character. A floating-point number will be rounded if it must be shortened to conform to a precision specification. The following program illustrates the use of precision specification with the floating-point numbers.

```
#include<stdio.h>
main()
{ float y=123.456789;
  clrscr();
  printf("%7f  %12f  %10.6f  %10.2f  %10.5f",y,y,y,y,y);
  getch();
}
```

When this program is run, the following output will be generated.

```
123.456787   123.456787 123.456787     123.46  123.45679
```

Minimum field width and precision specifications can be applied to character data as well as numeric data. When applied to string, the minimum field width is interpreted in the same manner as with a numerical quantity, i.e., leading blanks will be added if the string is shorter than the specified field width, and additional space will be allocated if the string is longer than the specified width. Hence, the field width specification will not prevent the entire string from being displayed. However, the precision specification will determine the maximum number of characters that can be displayed. If the precision specification is less than the total number of characters in the string, the excess right-most characters will not be displayed. This will occur even if the minimum field width is larger than the entire string, resulting in addition of leading blanks to the truncated string.

The following C program illustrates the use of minimum field width and precision specification in conjunction with string outputs.

```
#include<stdio.h>
main()
{ char text[80];
  clrscr();
  scanf("%[^\n]", text);
  printf("%3s %8s %10s %10.2s %.5s",text,text,text,text,text);
  getch();
}
```

Now suppose that the string Computer is entered from the keyboard when the program is executed. Then the following output will be generated.

```
Computer Computer   Computer        Co Compu
```

The first string is shown entirely, even though this string consist of 8 characters but the field width specification is only 3 characters. Thus, the first string overrides the minimum field width specification. The second string is displayed with only as many characters it has. The third string is padded with two leading blanks to fill out the 10-character minimum. Hence, the second string is right justified within its field. The fourth string consists of only two nonblank characters because of the 2-character precision specification. However, 8 leading blanks are added to fill out the minimum field width specification. The last string consists of 5 nonblank characters. Leading blanks are not added, however, because there is no minimum field width specification.

Certain conversion characters within the format string can be preceded by a single-letter prefix, which indicates the length of the corresponding argument. The allowable prefixes are same as the prefixes used with the scanf function. Thus, an l indicates a signed or unsigned integer argument, or a double precision argument; an h indicates a signed or unsigned short integer.

3.6. The gets() and put(s) Functions

The gets() and puts() functions accepts a single argument. The argument must be a data item that represents a string. The gets() function is used to read a string from a standard input device. The gets() function will be terminated by a newline character. The newline character will not be included as part of the string. The string may include white space characters. The string will be terminated by a null character.

The puts() function is used to display a string on a standard output device. The puts() function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with puts() is on its own line.

The usage of gets() and puts() functions are illustrated in the following program.

```
#include<stdio.h>
main()
{ char name[20];
  printf("Please enter your name : ");
  gets(name);
  printf("Hello ");
  puts(name);
}
```

Questions

3 Mark Questions

1. List and explain the commonly used input and output function in C.
2. What is the purpose of the getchar() function? How is it used within a program?
3. What are the disadvantages of using getchar() function in an interactive environment? Explain with an example.
4. How can getchar() function be used to read multicharacter strings. Give example.
5. What is purpose of getch() and getche() functions. Compare these functions with getchar() function.
6. What is the purpose of putchar() function? How is it used within a C program?
7. How can putchar() function be used to write multiple characters?
8. What is the purpose of the scanf function? How is it used within a program?
9. What is the purpose of the format string in scan function? What type information does it convey?
10. How is each character group within the format string identified? What are the constituent characters within a character group?
11. If a format string within a scanf contains multiple character groups, how are the character groups separated? Are whitespace characters required?
12. How are whitespace characters in the format string of a scanf function interpreted?
13. List and explain the commonly used conversion characters used within the format string of a scanf function.
14. How are multiple data items separated from one another while entering data to the scanf function?
15. When entering a string via the scanf function using s conversion character, how is the string terminated?
16. When entering a string via the scanf function how can a single string, which includes whitespace characters, be entered?
17. What is a field? How can the maximum field width specification for a data item be specified within a scanf function?
18. What is meant by maximum field width specification? How does it differ from the minimum field width specification?
19. How can short integer, long integer and double precision arguments be indicated within the format string of a scanf function?

20. What is assignment suppression? How can the assignment of an input data item to its corresponding argument be suppressed?
21. What is the purpose of printf function? How is it used within a C program?
22. Compare and contrast the scanf and printf functions.
23. What is the difference between f-type, e-type and g-type conversions when outputting floating-point data with printf function?
24. Compare the use of s-type conversion in the printf and scanf functions.
25. What is meant by minimum field width specification? How can minimum field width for a data item be specified within the printf function?
26. What is meant by precision of an output data item? How can precision be specified within printf function?
27. Explain the function of gets and puts function with an example.

8 Mark Question

1. Describe the commonly used console Input/Output functions in C.

Programming Questions

Draw flowchart and write C programs for the following

1. To input speed per hour and elapsed time and to calculate and display the distance travelled.
2. To input three integer quantities to the variables a, b, c and then perform the following
    a. Halves the value of a
    b. Doubles b
    c. Multiply c by itself
    d. Display the results of the preceding operations
3. To accept the length, breadth and height of a rectangular box and to display its volume.
4. To accept the length and breadth of a rectangle and to display its perimeter and area.
5. To accept the radius of a circle and to display it circumference ($2\pi r$) and area (πr^2).
6. To accept the radius of a sphere and to display its volume ($(4/3)\pi r^3$) and surface area ($4\pi r^2$).
7. To accept a number and to display the number along with its square, cube and square root.

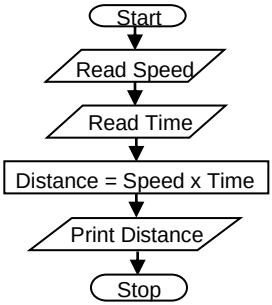
8. To accept a temperature in degree Celsius (C) and to display it in degree Fahrenheit (F) using the formula F =9.0/5.0×c+32
9. To convert a given number of days to a measure of time given in years, weeks, and days. For example, 375 days equals 1 year, 1 week, and 3 days. (Ignore leap year).
10. To Accept and add up five amounts and display the result to the nearest amount in rupees.
11. To round off a floating-point number to the nearest integer. Hint: add 0.5 to the number and truncate)
12. To round off a floating-point number to its nearest tenths and hundreds.
13. To round off an integer to its nearest tenth and hundredth.
14. To accept the principal amount (P), rate of interest (R) and number of years (N) and calculate and display the simple and compound interests (Simple interest = PNR, Compound interest = P-A, where $A=P(1+I)^n$)
15. To accept an amount in rupees and to calculate and display the number of currency notes for each denomination 1000, 500, 100, 50, 20, 10, 5, 2, 1, such that the total number of currency notes will be at minimum. Also output the total number of currency notes.
16. To accept a four digit number and to display the sum of its individual digits.
17. To compute mean and standard deviation of three numbers a, b and c. (Mean, m=(a+b+c)/3 and SD = $\sqrt{((a-m)^2+(b-m)^2+(c-m)^2)}/3$.
18. In an equation for a triangle, the sides a,b and c and the angle θ are related by the expression $b^2+c^2-a^2=2bCos(\theta)$. Input the length of the sides b and c and the angle θ and calculate and display the length of the side a.
19. Total mechanical energy of a projectile is given by $E=mgh+(mv^2/2)$, where m = mass, g = acceleration due to gravity, h = height and v = velocity. Accept the values of m, g, h and v. Calculate and display the energy.
20. A departmental store places an order with a company for m pieces of Electric Mixer, n pieces of Bread Toaster and p pieces of Electric Fans. The cost each item is

Mixer:	Rs. a per piece,
Bread Toaster:	Rs. b per piece
Electric Fan:	Rs. c per piece

    The discount allowed for various items are: 10% Mixer, 15% for Fans and 12.5% for Toaster. Accept data for m, n, p, a, b and c. Calculate and display the total cost to be paid by the store.

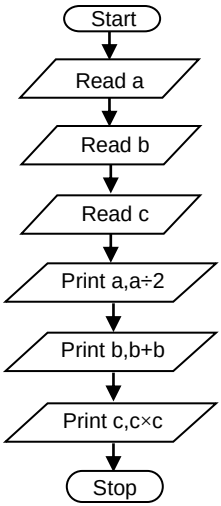
Answers to Select Problems

1. /* Program to calculate the distance travelled */




```
#include<stdio.h>
main()
{
      float distance, time, speed;
      clrscr();
      printf("Speed in Km/Hr ? ");
      scanf("%f",&speed);
      printf("Time of Journey ? ");
      scanf("%f",&time);
      distance = speed*time;
      printf("%.2f Kms travelled within %.2f Hrs at a speed of %.2f Km/Hr.",
      distance, time, speed);
      getch();
}
```

2.

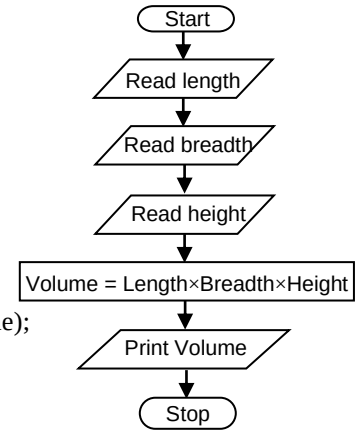



```
#include<stdio.h>
main()
{
  int a,b,c;
  clrscr();
  printf("Enter Three Numbers\n");
  printf("First Number  ? ");
  scanf("%d",&a);
  printf("Second Number ? ");
  scanf("%d",&b);
  printf("Third Number  ? ");
  scanf("%d",&c);
  printf("Half of %d is %g\n",a,a/2.0);
  printf("Double of %d is %d\n",b,b+b);
  printf("Square of %d is %d\n",c,c*c);
  getch();
}
```

3.

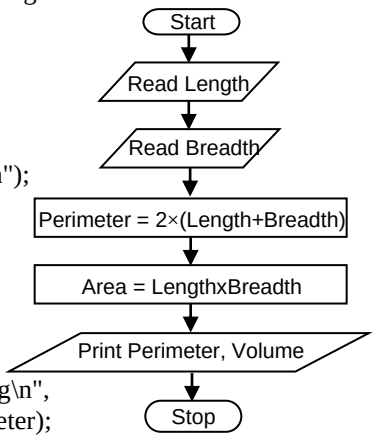
```
/* To calculate Volume of a box */
#include<stdio.h>
main()
{
```

```
  float length, breadth, height,volume;
  clrscr();
  printf("Enter Dimension of a box\n");
  printf("Length  ? ");
  scanf("%f",&length);
  printf("Breadth ? ");
  scanf("%f",&breadth);
  printf("Height  ? ");
  scanf("%f",&height);
  volume = length*breadth*height;
  printf("Volume of the box is %.3g",volume);
  getch();
}
```



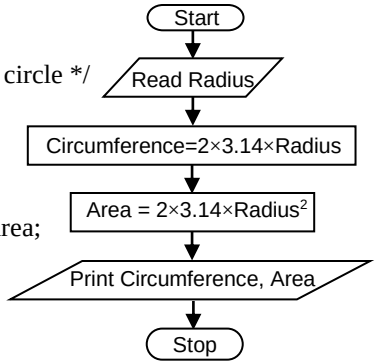
4.

```
/* To calculate Perimeter & Area of a rectangle */
#include<stdio.h>
main()
{
  float length, breadth, perimeter, area;
  clrscr();
  printf("Enter Dimension of a rectangle\n");
  printf("Length  ? ");
  scanf("%f",&length);
  printf("Breadth ? ");
  scanf("%f",&breadth);
  perimeter = 2*(length+breadth);
  area = length*breadth;
  printf("Perimeter of the rectangle is %.3g\n",
                                    perimeter);
  printf("Area of the rectangle is %.3g",area);
  getch();
}
```



5.

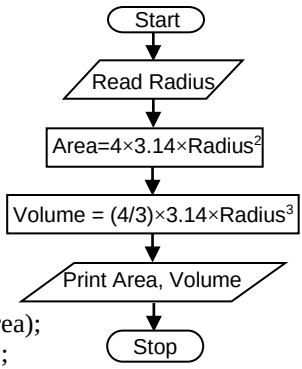
```
/* To calculate circumference & area of a circle */
# define pi 3.14
#include<stdio.h>
main()
{
  float radius, circumference, perimeter, area;
  clrscr();
  printf("Enter Radius of the circle ");
  scanf("%f",&radius);
  circumference = 2*pi*radius;
```



```
  area = pi*radius*radius;
  printf("Circumference of the circle is %.3g\n",
                                    circumference);
  printf("Area of the circle is %.3g",area);
  getch();
}
```

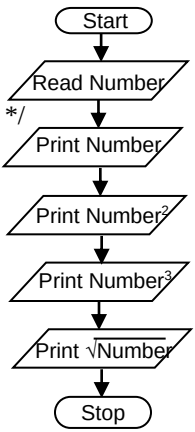
6.

```
/* To calculate surface area and volume of a sphere */
# define pi 3.14
#include<stdio.h>
main()
{
  float radius, area, volume;
  clrscr();
  printf("Enter Radius of the sphere ");
  scanf("%f",&radius);
  area = 4*pi*radius*radius;
  volume = (4.0/3.0)*pi*pow(radius,3);
  printf("Surface area of the sphere is %.3g\n",area);
  printf("Volume of the sphere is %.3g",volume);
  getch();
}
```



7.

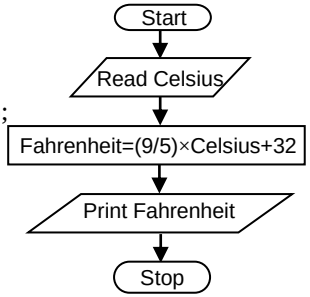
```
/* To display a number, its square, its cube and square root */
#include<stdio.h>
# include<math.h>
main()
{
  float number;
  clrscr();
  printf("Enter a Number: ");
  scanf("%f",&number);
  printf("The number you entered is %g\n",number);
  printf("Square of %g is %g\n",number,pow(number,2));
  printf("Cube of %g is %g\n",number,pow(number,3));
  printf("Square root of %g is %g\n",number,sqrt(number));
  getch();
}
```



8.

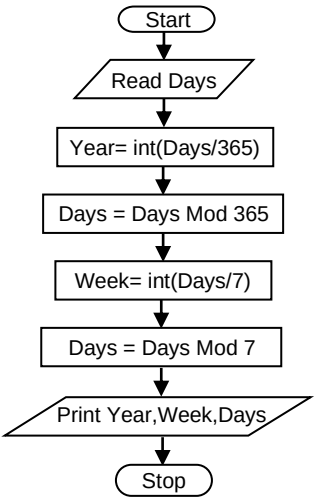
```
/* To convert temperature in degree Celsius to Fahrenheit */
#include<stdio.h>
main()
```

```
{
   float celsius, fahrenheit;
   clrscr();
   printf("Enter temperature in degree Celsius : ");
   scanf("%f",&celsius);
   fahrenheit = (9.0/5.0)*celsius+32;
   printf("%g Degree celsius is %g Fahrenheit",
                                  celsius,fahrenheit);
   getch();
}
```

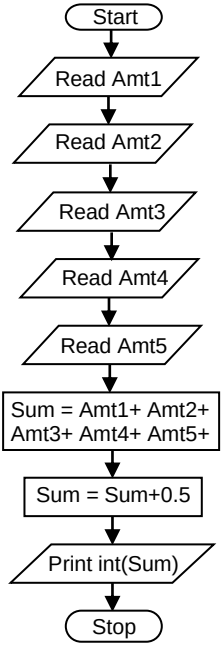


9.

```
/* To convert number of days into years, weeks and days */
#include<stdio.h>
main()
{
   int temp,days,year,week,day;
   clrscr();
   printf("Enter the number of days : ");
   scanf("%d",&days);
   temp = days;
   year = (int)days/365;
   days = days%365;
   week = (int)days/7;
   days = days%7;
   printf("%d days equals %d year, %d week and %d days"
                        ,temp,year,week,days);
   getch();
}
```

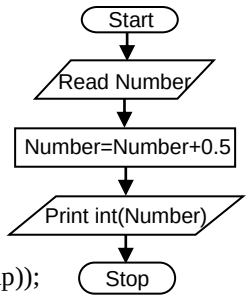


10.



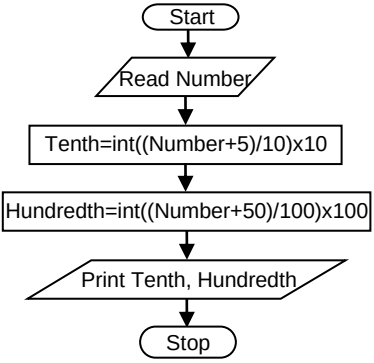

```
/* To read and sum up 5 amounts in rupees
   and round the amount to the nearest rupee */
#include<stdio.h>
#include<math.h>
main()
{
  float amt1,amt2,amt3,amt4,amt5,sum,temp;
  clrscr();
  printf("Enter Five amounts in Rupees : \n");
  printf("First Amount   ? ");
  scanf("%f",&amt1);
  printf("Second Amount  ? ");
  scanf("%f",&amt2);
  printf("Third Amount   ? ");
  scanf("%f",&amt3);
  printf("Fourth Amount  ? ");
  scanf("%f",&amt4);
  printf("Fifth  Amount  ? ");
  scanf("%f",&amt5);
  sum = amt1+amt2+amt3+amt4+amt5;
  printf("Total of %.2f, %.2f, %.2f, %.2f and %.2f is %.2f\n",
                          amt1,amt2,amt3,amt4,amt5,sum);
  temp=sum+0.5;
  printf("Rs. %.2f can be rounded to Rs. %g",sum,floor(temp));
  getch();
}
```

11.



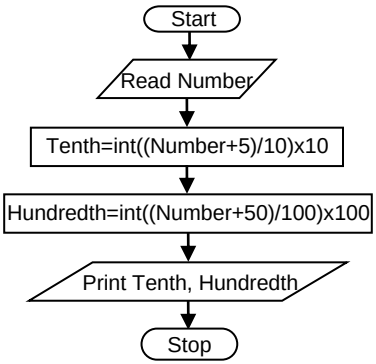

```
#include<math.h>
main()
{
  float number,temp;
  clrscr();
  printf("Enter the Number ");
  scanf("%f",&number);
  temp=number+0.5;
  printf("%g can be rounded to %g",number,floor(temp));
  getch();
}
```

12.



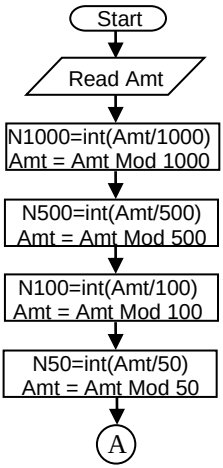

```
/* To round a floating-point number
    to its nearest tenth and hundredth */
#include<stdio.h>
main()
{
  float number;
  int tenth, hundredth;
  clrscr();
  printf("Enter the Number ");
  scanf("%f",&number);
  tenth=((int)(number+5)/10)*10;
  hundredth=((int)(number+50)/100)*100;
  printf("%g can be rounded to the nearest tenth as %d\n",number,tenth);
  printf("%g can be rounded to the nearest hundredth as %d" ,
          number,hundredth);
  getch();
}
```

14.



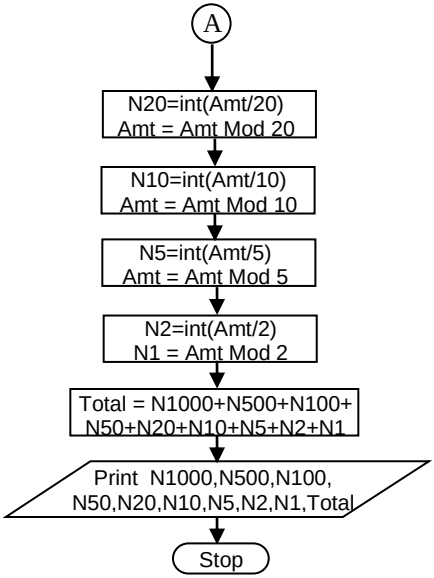

```
/* To round an integer to its
    nearest tenth and hundredth */
#include<stdio.h>
main()
{
  int number,tenth,hundredth;
  clrscr();
  printf("Enter the Number ");
  scanf("%d",&number);
  tenth=((number+5)/10)*10;
  hundredth=((number+50)/100)*100;
  printf("%d can be rounded to the nearest
       tenth as %d\n",number,tenth);
  printf("%d can be rounded to the nearest
       hundredth as %d",number,hundredth);
  getch();
}
```

15.




```
/* To find the denomination of an amount*/
#include<stdio.h>
main()
```

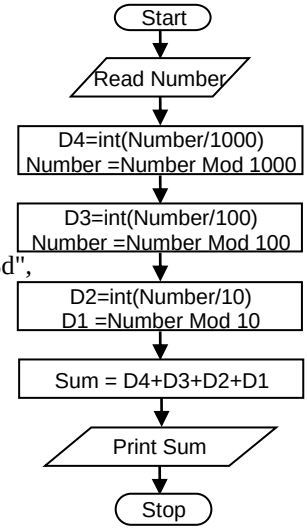
```
{
  int amt,temp,n1000,n500,n100;
  int n50,n20,n10,n5,n2,n1,total;
  clrscr();
  printf("Enter the Amount ");
  scanf("%d",&amt);
  temp = amt;
  n1000 = amt/1000;
  amt = amt%1000;
  n500 = amt/500;
  amt = amt%500;
  n100 = amt/100;
  amt = amt%100;
  n50 = amt/50;
  amt = amt%50;
  n20 = amt/20;
  amt = amt%20;
  n10 = amt/10;
  amt = amt%10;
  n5 = amt/5;
  amt = amt%5;
  n2 = amt/2;
  n1 = amt%2;
  total = n1000+n500+n100+n50+n20+n10+n5+n2+n1;
  printf("Rs. %d contains %d 1000 Rupee notes\n",temp,n1000);
  printf("Rs. %d contains %d  500 Rupee notes\n",temp,n500);
  printf("Rs. %d contains %d  100 Rupee notes\n",temp,n100);
  printf("Rs. %d contains %d   50 Rupee notes\n",temp,n50);
  printf("Rs. %d contains %d   20 Rupee notes\n",temp,n20);
  printf("Rs. %d contains %d   10 Rupee notes\n",temp,n10);
  printf("Rs. %d contains %d    5 Rupee notes\n",temp,n5);
  printf("Rs. %d contains %d    2 Rupee notes\n",temp,n2);
  printf("Rs. %d contains %d    1 Rupee  notes\n",temp,n1);
  printf("Rs. %d contains a total of %d notes",temp,total);
  getch();
}
```



16.

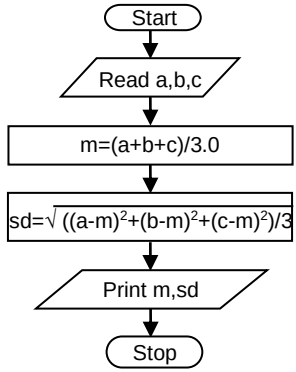
```
/* To display the sum of the individual digits
   of a four digit number */
#include<stdio.h>
main()
{  int number,temp,d1,d2,d3,d4,sum;
   clrscr();
```

```
  printf("Enter a Number ");
  scanf("%d",&number);
  temp = number;
  d4 = number/1000;
  number = number%1000;
  d3 = number/100;
  number = number%100;
  d2 = number/10;
  d1 = number%10;
  sum = d4+d3+d2+d1;
  printf("Sum of the individual digits of %d is %d",
                                        temp,sum);
  getch();
}
```



17.

```
/* To display the mean and standard deviation
   of three numbers */
#include<stdio.h>
#include<math.h>
main()
{
  float a,b,c,m,sd;
  clrscr();
  printf("Enter a Three Numbers\n");
  printf("First Number ? ");
  scanf("%f",&a);
  printf("Second Number ? ");
  scanf("%f",&b);
  printf("Third Number ? ");
  scanf("%f",&c);
  m=(a+b+c)/3.0;
  sd = sqrt((pow((a-m),2)+pow((b-m),2)+pow((c-m),2))/3.0);
  printf("Average of %g, %g and %g is %g\n",a,b,c,m);
  printf("Standard deviation of %g, %g and %g is %g",a,b,c,sd);
  getch();
}
```



UNIT IV

CONTROL STATEMENTS

4.1. Introduction

Control statements are used to create special program features, such as logical test, loops and branches. There are three types of control statements in C. They are,

1. Selection statement or Conditional statements, such as if and switch.
2. Iteration statements or Loop statements, such as while, for and do-while.
3. Jump statements such as break, continue, goto and return.

4.2. Selection Statements

A selection statement selects a group of statements from several available groups, depending on the outcome of a logical test. C supports two selection statements: if and switch.

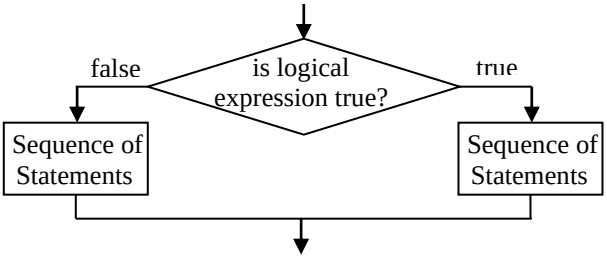
The if Statement

The if statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false). The general form of the if statement is

```
if(expression)
    statement1;
else
    statement2;
```

where the statement1 or statement2 can be either simple or compound statement. The else clause is optional. The expression must be placed in parentheses. If the expression has a nonzero value (i.e., if the expression is true) then statement1 will be executed. Otherwise (i.e., if the expression is false), statement2 will be executed.

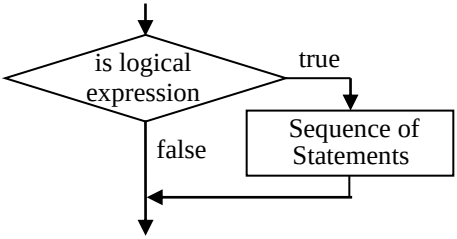
This form of if a statement is represented in the following flowchart



The else portion of the if is optional. Thus in its simplest form, the statement can be written as

if (expression) statement;

In this form, the statement will be executed only if the expression has a nonzero value (i.e., if the expression is true). If the expression has a value of zero (i.e., if the expression is false), then the statement will be ignored.
This form of if a statement is represented in the following flowchart



Several representative if statements are shown below.

```
if(x<0) printf("%f",x);

if(p>0)
      credit = 0;

if(x<=3.0)
      {  y=3*pow(x,2);
         printf("%f\n",y);
      }

if((balance<1000.0)||(status=='R'))
         printf("%f",balance);

if((a>=0)&&(b<=5)
      {  xmid= (a+b)/2;
         ymid= sqrt(xmid);
      }

if(status=='S')
      tax=0.02*pay;
else
      tax=0.14*pay;
```

```
if(p>0)
      {  printf("Overdue");
         credit = 0;
      }
else
         credit=1000.0;

if(x<=3.0)
         y=3*pow(x,2);
else
         y=2*pow((x-3),2);
printf("%f\n",y);

if (circle)
      {  scanf("%f",&radius);
         area=3.14*radius*radius;
         printf(" Area of circle is ",
         area);
      }
else
      {  scanf("%f  %f",
         &length,&breadth);
         area=length*breadth;
         printf("Area of rectangle is "
         ,area);
      }
```

It is possible to nest (i.e., embed) if-else statements, one within another. The if statements can be nested in several different forms. The most general form of two-layer nesting is

```
if(e1)  if (e2)
            s1;
        else
            s2;
else
        if (e3)
            s3;
        else
            s4;
```

where e1,e2 and e3 represent logical expressions and s1,s2,s3 and s4 represent statements. One complete if-else statement will be executed if e1 is nonzero (true), and another complete if-else statement will be executed if e1 is zero (false).

Some other forms of two-layer nesting are

```
if(e1)
   s1;
else
   if(e2)
      s2;
```

```
if(e1)
   s1;
else
   if(e2)
      s2;
   else
      s3;
```

```
if(e1)
   if(e2)
      s1;
   else
      s2;
else
      s3;
```

```
if(e1)
   if(e2)
      s1;
   else
      s2;
```

While evaluating the nested if statement the rule is that the else clause is always associated with closest preceding unmatched (i.e., else-less) if.

In some situations it may be desirable to nest multiple if-else statements, in order to create a situation in which one of several different courses of action will be selected. Its general form is

```
if(e1)
   s1;
else
   if(e2)
      s2;
   else
      if(e3)
         s3;
      else
         .
         .
         .
         sn;
```

The conditions are evaluated from top to downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final else is executed. That is, if all other conditional test fail, the last else statement is performed. If the final else is not present, no action takes place if all other conditions are false.

The switch Statement

Switch is a multiple-branch selection control statement in which one group of statement is selected from several available groups. The selection is based upon the current value of an expression, which is included within the switch statement. The general form of the switch statement is

switch(expression) statement;

where expression results in an integer value. The expression can also be of type char, since individual characters have equivalent integer values.

The embedded statement is generally a compound statement that specifies alternate courses of action. Each alternative is expressed as a group of one or more individual statements within the overall embedded statement. For each alternative, the first statement within the group must be preceded by one or more case labels. The case labels identify the different groups of statements and distinguish them from one another. The case labels must therefore be unique within a given switch statement. The general form of each group of statement is

```
case expression :
statement1;
statement2;
...
statementn;
```

or, when multiple case labels are required,

```
case expression1 :
case expression2 :
...
case expressionm :
statement1;
statement2;
...
statementn;
```

where expression1, expression2, ..., expressionm represent constant, integer-valued expressions. Usually, each of these expressions will be written as

either as an integer constant or a character constant. Each individual statement following the case label may be either simple or compound.
Thus, the general form of the switch statement is

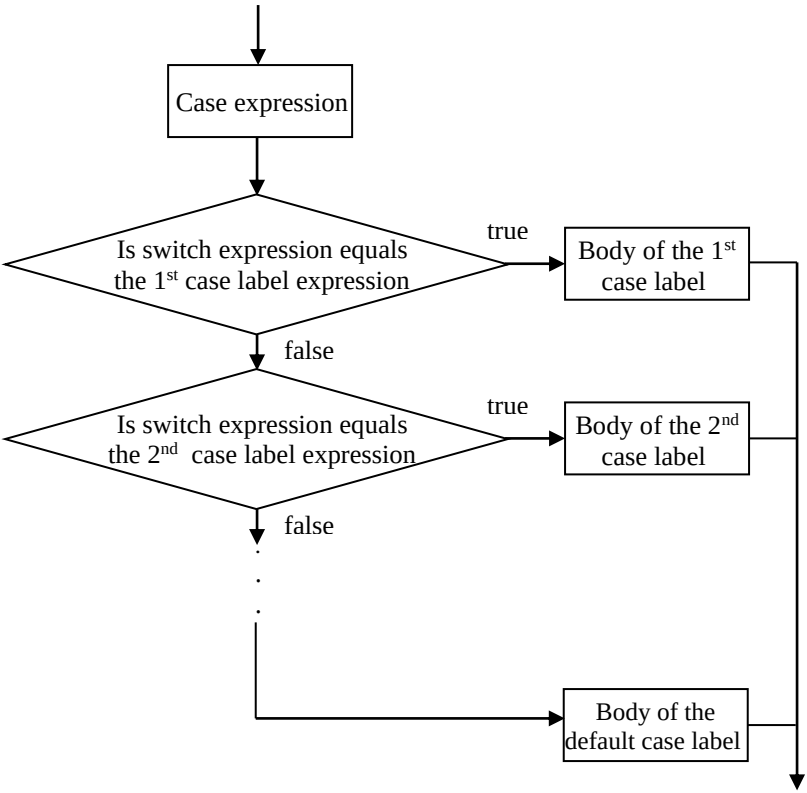
```
switch(expression)
{  case expression_1_1:
   case expression_1_2:
   case expression_1_3:
   . . .
   case expression_1_n:
        statement_1;
        statement_2;
        . . .
        statementn
   case expression_2_1:
   case expression_2_2:
   case expression_2_3:
   . . .
   case expression_2_n:
        statement_1;
        statement_2;
        . . .
        statement_n
   . . .
   case expression_m_1:
   case expression_m_2:
   case expression_m_3:
   . . .
   case expression_m_n:
        statement_1;
        statement_2;
        . . .
        statement_n
   default:
        statement_1;
        statement_2;
        . . .
        statement_n
}
```

When the switch statement is executed, the expression in the switch statement is evaluated and the control is transferred directly to the group of statements whose case label value matches the value of the expression. If none of the case-label values matches the value of the expression, then none

of the groups within the switch statement will be selected and the control is transferred directly to the statement that follows the switch statement.

One of the labelled groups of statements within the switch statement may be labelled default. This group will be selected if none of the case labels matches the value of the expression. The default group may appear anywhere within the switch statement – it need not necessarily be placed at the end. If none of the case labels matches the value of the expression and the default group is not present, then no action will be taken by the switch statement.

The following flowchart illustrates the operation of the switch statement



The following program illustrates the use of switch statement.

```
/* To input the first letter of the colours in VIBGYOR and
to display the name of the colour */
#include<stdio.h>
main()
{  char c;
```

```
    clrscr();
    printf("Enter any letter from VIBGYOR : ");
    switch(c=getchar())
    {  case 'v':
       case 'V': printf("Violet");
                 break;
       case 'i':
       case 'I':  printf("Indigo");
                 break;
       case 'b':
       case 'B': printf("Blue");
                 break;
       case 'g':
       case 'G': printf("Green");
                 break;
       case 'y':
       case 'Y': printf("Yellow");
                 break;
       case 'o':
       case 'O': printf("Orange");
                 break;
       case 'r':
       case 'R': printf("Red");
                 break;
       default : printf(" Bad data");
    }
}
```

The switch statement in the above can be written more concisely as

```
    switch(toupper(c=getchar()))
    {  case 'V': printf("Violet");
                 break;
       case 'I':  printf("Indigo");
                 break;
       case 'B': printf("Blue");
                 break;
       case 'G': printf("Green");
                 break;
       case 'Y': printf("Yellow");
                 break;
       case 'O': printf("Orange");
                 break;
       case 'R': printf("Red");
                 break;
```

```
    default : printf(“ Bad data”);
}
```

Here multiple case labels for each group of statements are avoided by the use of the library function toupper.

The break statement is used to exit from the switch. The break statement causes a transfer of control out of the entire switch statement, to the first statement following the switch statement.

The switch statement can be nested by having a switch as a part of the statement sequence of an outer switch. Even if the case expressions of the inner and outer switch contain common values, no conflicts arise. For example, the following code fragment is perfectly acceptable.

```
switch(x)
{
  case 1:
    switch(y)
    {
      case 0:  printf(“Divide by zero error\n”);
               break;
      case 1:  process(x,y)
               break;
    }
  case 2:
    . . .
```

The switch statement can be considered a good alternative to the nested if-else statements that test for equality. The switch differs from the if statement in that switch can only test for equality, whereas if can evaluate any type of relational or logical expression.

4.3. Iteration Statements

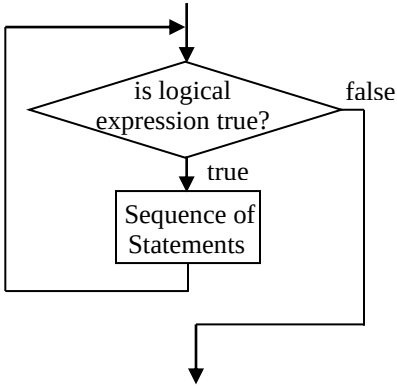
The while Statement

The while statement is used to carry out looping operations, in which a group of statement is executed repeatedly, until some condition has been satisfied. The general form of while statement is

```
while (expression) statement;
```

The statement will be executed repeatedly, as long as the expression is true (i.e., as long as the expression has a nonzero value). The statement can be either simple or compound. The statement must include some feature that eventually alters the value of the expression, thus providing stopping condition for the loop.

The following flowchart illustrates the functioning of the while loop.



The following program illustrates the use of the while statement.

```
/* To display the whole number up to 100*/
#include<stdio.h>
main()
{     int number=0;
      clrscr();
      while(number<=100)
      {     printf("%d  ",number);
            number++;
      }
  getch();
}
```

This program can be written more concisely as

```
/* To display the whole number up to 100*/
#include<stdio.h>
main()
{     int number=1;
      clrscr();
      while(number<=100)
            printf("%d  ",number++);
      getch();
}
```

The while statement is particularly useful when the number of passes through the loop is not known in advance. In such situations, the

looping action needs to be continued indefinitely, as long as the specified logical condition is satisfied.

The following example illustrates the use while statement in a situation where the exact number of passes through the loop is unknown.

```
/* To convert a lowercase text into uppercase */
#include<stdio.h>
main()
{     int count=0;
      char text[80];
      clrscr();
      text[count]=getchar();
      while(text[count]!= '\n')
      {  count++;
         text[count]=getchar();
      }
      count = 0;
      while(text[count]!='\n')
      {  putchar(toupper(text[count]));
         count++;
      }
getch();
}
```

This program can be written more concisely as

```
/* To convert a lowercase text into uppercase */
#include<stdio.h>
main()
{     int count=0;
      char text[80];
      clrscr();
      while((text[count++]=getchar()) != '\n');
      count = 0;
      while(text[count]!='\n')
            putchar(toupper(text[count++]));
      getch();
}
```

In while statement, the test for continuation of the loop is carried out at the beginning of each pass. Therefore, the statement part will not be executed even once if condition is initially false. The while statement is well suited if the number of passes through the loop is not known in advance and the looping action is to be continued indefinitely until the specified logical condition is satisfied.

The do-while Statement

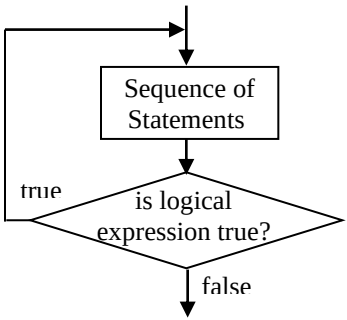
The do-while statement is also used to carry out looping operations, in which a group of statement is executed repeatedly, until some condition has been satisfied. The general form of while statement is

do statement while (expression);

The statement will be executed repeatedly, as long as the value of the expression is true (i.e., nonzero). The statement can be either simple or compound. The statement must include some feature that eventually alters the value of the expression, thus providing stopping condition for the loop.

In do-while statement, the test for continuation of the loop is carried out at the end of the each pass. Therefore, the statement part will be always executed at least once even if the condition is initially false. The do-while statement is well suited if the number of passes through the loop must be at least one and the exact number passes through the loop is not known in advance.

The following flowchart illustrates the functioning of the while loop.



The following example illustrates the use of do-while loop.

```
/* display the integers 0 through 100 */
#include<stdio.h>
main()
{  int digit = 0;
   do
      printf("%d  ",digit++);
   while(digit<=100)
}
```

The following example illustrates that the use of do-while loop is preferred over the while loop if the number passes through the loop is at least one.

```
/* To display the sum and average of a set of numbers.
   The program should terminate only when a zero is entered */
#include<stdio.h>
main()
{    int count=0;
     float number,sum=0.0,average;
     clrscr();
      do
     {  printf("Enter your No-%d (Zero to Stop) : " ,++count);
        scanf("%f",&number);
        sum = sum + number;
      }
     while(number != 0);
     average = sum/(count-1);
     printf("Sum is %g\nAverage is %g",sum,average);
     getch();
}
```

The for Statement

The for statement is also used to carry out looping operations, in which a group of statement is executed repeatedly for a specific number of times. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of for statement is

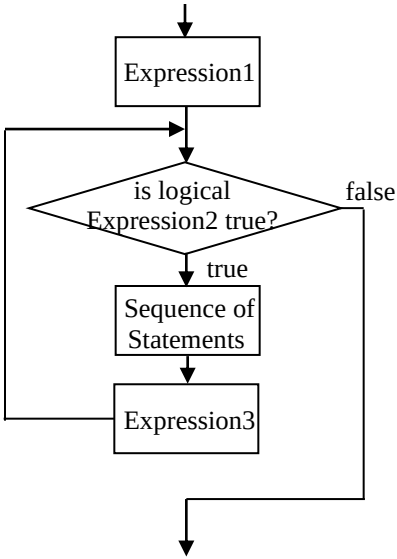
for(expression1;expression2;expression3) statement;

The for loop allows many variations, but its most common form work like this: The expression1 is used to initialise some parameter (called index) that controls the looping action, expression2 represents a logical condition that must be true for the loop to continue execution, and expression3 is used to alter the value of the index initially assigned by expression1. Typically, expression1 is assignment expression, expression2 is a logical expression and expression3 is a unary expression or an assignment expression.

When the for statement is executed, expression2 is evaluated and tested at the beginning of each pass through the loop, and expression3 is evaluated at the end of each pass. Thus, the for statement is equivalent to

```
expression1;
while(expression2)
{    statement;
     expression3;
}
```

The looping action will continue as long as the value of the expression2 is not zero, i.e., as long as the logical condition represented by expression2 is true. The operation of the for loop is illustrated in the following flowchart.



The for statement, like the while and do-while statements, can be used to carry out looping actions where the number passes through the loop is not known in advance. Because of the features that are built into the for statement, however, it is particularly well suited for loops in which the number of passes is known in advance.
The following example illustrates the use of for statement.

```
/* To display the integers 0 through 100 */
#include<stdio.h>
main()
{     int digit;
      for(digit=0;digit<=100;++digit)
         printf("%d  ",digit);
}
```

The expression1 and expression3 can be omitted from the syntax of the for statement, if other means are provided for initialising the index and/or altering the index. This is illustrated in the following example.

```
/*To display the integers 0 through 100 */
#include<stdio.h>
main()
{  int digit=0;
   for(;digit<=100;)
```

```
        printf(“%d”,digit++);
}
```

However, the semicolon must be present. If expression2 is omitted, however, it will be assumed to have a permanent value of 1 (true) and therefore the loop will continue indefinitely unless it is terminated by some other means, such as a break or return statement. This is illustrated in the following example.

```
/* To display the sum and average of a set of numbers.
   The program should terminate only when a zero is entered */
#include<stdio.h>
main()
{  int count=0;
   float number,sum=0.0,average;
   clrscr();
   for(;;)
   {  printf("Enter your No-%d (Zero to Stop) : " ,++count);
      scanf("%f",&number);
      if (number == 0) break;
      sum = sum + number;
   }
   average = sum/(count-1);
   printf("Sum is %g\nAverage is %g",sum,average);
   getch();
}
```

The use of comma operator (,) within for statements permits two different expressions to appear in situations where only one expression would ordinarily be used. For example, it is possible to write

```
for(expression1a, expression1b; expression2; expression3a, expression3b)
```

where expression1a and expression1b are two expressions, separated by comma operator, where only one expression (expression1) would normally appear. Similarly, expression3a and expression3b, separated by comma operator, appear in place of the usual single expression. The use of comma operator in the for statement is illustrated in the following C program.

```
/* To display the series 0,100,1,99,2,98,...,99,1,100,0 */
#include<stdio.h>
main()
{ int a,b;
  clrscr();
  for(a=0,b=100;a<=100;++a,--b)
     printf("%d\t%d\t",a,b);
  getch();
}
```

The comma operator accepts two distinct expressions as operands. Several comma operators can be used to string together several expressions. These expressions will be evaluated from left to right. In situations that require the evaluation of the overall expression (i.e., the expression formed by the two operands and the comma operator), the type and the value of the overall expression will be determined by the type and value of the right operand. Within the collection of C operators, the comma operator has the lowest precedence. Its associativity is from left to right. This is illustrated in the following C program

```
/* To illustrate the precedence and associativity of comma operator */
#include<stdio.h>
main()
{ int a,b;
  clrscr();
  a = (1,2,3,4,5,6,7,8,9,10);
  b=(a,a+2,10,a/2,a+2*20);
  printf("a=%d\tb=%d",a,b);
  getch();
}
```

The assignment statement a = (1,2,3,4,5,6,7,8,9,10) first assigns a the value 1, then the value 2 and so on. Finally, a is assigned the value of 10. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator. The second assignment statement causes the value of the expression a+2*20, i.e., 50, to be assigned to the variable b. Thus, this program will output result as a=10 and b=50.

4.4. Jump Statements

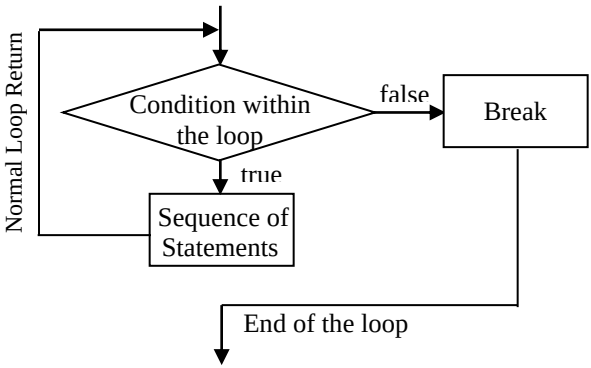
The break Statement

The break statement is used to terminate loops or to exit from a switch. It can be used within a for, while, do-while or switch statement. The break statement is simply is written as

```
break;
```

without any embedded expressions or statements. In switch statement, the break statement causes a transfer of control out of the entire switch statement, to the first statement following the switch statement. If a break statement is included in while, do-while or for loop, the control will immediately be transferred out of the loop when the break statement is encountered. In the event of several nested while, do-while, for or switch

statement a break statement will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements. The following flowchart indicates the functioning of the break statement.



The following program illustrates the use of break statement in a while loop.

```
/* To display the sum and average of a set of numbers.
   The program should terminate only when a zero is entered */
#include<stdio.h>
main()
{  int count=0;
   float number,sum=0.0,average;
   clrscr();
   while(1)
   {  printf("Enter your No-%d (Zero to Stop) : " ,++count);
      scanf("%f",&number);
      if (number == 0) break;
      sum = sum + number;
   }
   average = sum/(count-1);
   printf("Sum is %g\nAverage is %g",sum,average);
   getch();
}
```

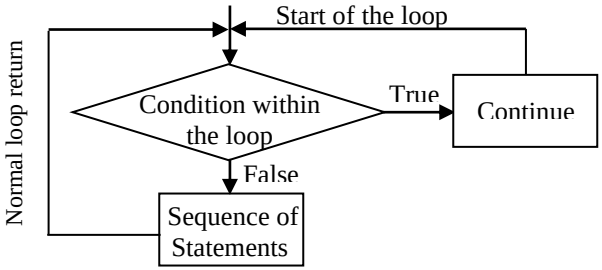
The continue Statement

The continue statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a continue statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. The

continue statement can be included within a while, do-while, or for statement. It is written simply as

continue;

without any embedded expressions or statements. The following flowchart illustrates the functioning of the continue statement.



The following program illustrates the use of continue statement.
/* To display the sum and average of nonnegative numbers from a set of given numbers. The program should terminate only when a zero is entered */

```
#include<stdio.h>
main()
{    int count=0;
     float number,sum=0.0,average;
     clrscr();
     while(1)
      { printf("Enter your No-%d (Zero to Stop) : " ,++count);
        scanf("%f",&number);
        if (number == 0) break;
        if (number < 0 ) continue;
        sum = sum + number;
     }
     average = sum/(count-1);
     printf("Sum is %g\nAverage is %g",sum,average);
     getch();
}
```

The goto Statement

The goto statement is used to alter the normal sequence program execution by transferring control to some other part of the program unconditionally. In its general form, the goto statement is written as

goto label;

where the label is an identifier that is used to label the target statement to which the control is transferred. Control may be transferred to anywhere within the current function. The target statement must be labelled, and the label must be followed by a colon. Thus, the target statement will appear as

label : statement;

Each labelled statement within the function must have a unique label, i.e., no two statements can have the same label.

All of the general-purpose programming languages contain a goto statement, though the modern programming practice discourages its use. The goto statement was used extensively, however, in early versions of some older languages, such as FORTRAN and BASIC. The most applications of goto statements were:

1. Branching around statements or group of statements under certain conditions.
2. Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass.
3. Jumping completely out of a loop under certain conditions, thus terminating the execution of a loop.

The structured features in C enables all these operations to be carried out without the use of goto statements. For example, branching around statements can be accomplished with the if-else statement; jumping to the end of a loop can be carried out with the continue statement; and jumping out of a loop is easily accomplished using the break statement. The use of these structured features is preferable to the use of the goto statement, because the use of goto tends to encourage (or at least not discourage) logic that skips all over the program where as the structured features in C requires that the entire program be written in an orderly, sequential manner. For this reason, the use of goto statement should generally be avoided

Occasional situation do arise, however, in which the got statement can be useful. Consider, for example, a situation in which it is necessary to jump out of a doubly nested loop if a certain condition is detected. This can be accomplished with two if-break statements, one within each loop, though this is awkward. This is illustrated in the following program segment.

```
/* To determine whether two array have an element on common */
        flag = 0;
        for(i=0;i<n;++i)
        {    for(j=0,i<m,++j)
             if(x[i]==y[j])
             {  flag=1;
                break;
             }
          if(flag) break;
```

```
        }
        if(flag) printf(" Identical Elements exists");
```

A better solution in this particular situation might make use of the goto statement to transfer of both loops at once. This is illustrated in the following program segment.

```
/* To determine whether two arrays have an element on common */
        flag = 0;
        for(i=0;i<n;++i)
           for(j=0,i<m,++j)
              if(x[i]==y[j])
              {  flag =1;
                 goto found;
              }
        found :
              if(flag) printf(" Identical Element exists");
```

Questions

3 Mark Questions

1. What are control statements? List the three different types of control statements with examples.
2. What are selection or conditional statements? Give examples.
3. What are loop or iteration statements? Give examples.
4. What is looping? Describe two different forms of looping.
5. What are jump statement? Give examples.
6. What is the purpose of the if statement? Summarize the syntactic rules associated with the if statement.
7. Describe the two different forms of if statement. How do they differ?
8. Explain the different forms of nesting if statements.
9. How is the nested if statement `if(e1) if (e2) s1; else s2;` is interpreted?
10. What is the purpose of the switch statement? Summarize the syntactic rules associated with the switch statement.
11. Compare the use switch statement with the use of nested if statement. Which is more convenient?
12. What is the purpose of default alternative in switch statement? Illustrate with an example.
13. Explain the functioning of a switch statement. Illustrate with an example.
14. What is the purpose of the while statement? Summarize the syntactic rules associated with the while statement.
15. Explain the functioning of while statement. Illustrate with an example.
16. What is the purpose of do-while statement? Summarize the syntactic rules associated with the do-while statement.
17. Differentiate between the execution of while and do-while statements. Explain their relative merits and demerits.
18. What is the minimum number of times that a do-while loop can be executed? Compare with a while loop and explain reasons for difference.
19. What is the purpose of the for statement? Summarize the syntactic rules associated with the for statement.
20. How many times will a for loop be executed? Compare it with the while loop and do-while loops.
21. What is the purpose of the break statement? Illustrate with an example.
22. What is the purpose of the continue statement? Illustrate with an example.

23. What is the function of comma operator? What is the precedence of comma operator compared with other C operators?
24. What is the purpose of the goto statement? Summarize the syntactic rules associated with the goto statement.
25. Why is the use of goto statement generally discouraged? Under what conditions might the goto statements be helpful? Illustrate with an example.

8 Mark Question

1. Describe the different control statements available in C.

Programming Questions

Draw flowchart and write C program for the following questions

1. To input a number and to display whether it is even or odd
2. To input a number and to display whether it is a multiple of 5 or not.
3. To input 3 numbers and to display the biggest among them.
4. To input 3 numbers and to display the biggest and smallest among them.
5. To display the whole numbers 1,2,3,...,100.
6. To display the even number 2,4,6,...,100.
7. To display the odd numbers 1,3,5,...,99.
8. To display the whole numbers 100,99,98,...,1.
9. To display the even number 100,98,96,...,2.
10. To display the odd numbers 99,97,95,...,1.
11. To display the whole numbers 1,2,3,... up to a given number.
12. To display the even number 2,4,6,... up to a given number.
13. To display the odd numbers 1,3,5,... up to a given number.
14. To display the even number 2,4,6,... up to the n^{th} term.
15. To display the odd numbers 1,3,5,... up to the n^{th} term.
16. To display the first 'n' whole numbers in reverse order.
17. To display the first 'n' odd numbers in reverse order.
18. To display the first 'n' even numbers in reverse order.
19. To display the whole numbers between two given numbers (within a range)
20. To display the odd numbers within a range.
21. To display the even numbers within a range.
22. To display the multiples 5 within a range.

23. To display the first ‘n’ whole numbers. Also display their sum and average.
24. To display the first ‘n’ even numbers. Also display their sum and average.
25. To display the first ‘n’ odd numbers. Also display their sum and average.
26. To display the whole numbers within a range. Also display their sum and average.
27. To display the odd numbers within a range. Also display their sum and average.
28. To display the even numbers within a range. Also display their sum and average.
29. To display the odd and even numbers within a range. Display the numbers separately. Also display their sum and average.
30. To check whether a given number is prime or not.
31. To display the prime numbers up to a given number.
32. To display the first ‘n’ prime numbers.
33. To display the prime numbers within a range.
34. To check whether a given number is Armstrong number or not. (The sum of the cubes of the individual digits of an Armstrong number will be equivalent to the number itself, e.g., $153=1^3+5^3+3^3$)
35. To display the Armstrong numbers up to a given number.
36. To display the first ‘n’ Armstrong numbers.
37. To display the Armstrong numbers within a range.
38. To check whether a given number is a fibonacci number or not.
39. To display the fibonacci numbers 1,1,2,3,5,8, ... up to a given number.
40. To display the first ‘n’ fibonacci numbers.
41. To display the fibonacci numbers within a range.
42. To display the factorial of a number.
43. To check whether a given number is a perfect factorial or not.
44. To sum the series 1!+2!+3!+....+10!.
45. To display the sum of the factorials of the odd numbers from 1 to 10.
46. To display the sum of the factorials of the even numbers 1 to 10.
47. To sum the series 1!+2!+3!+.... up to the n^{th} term.
48. To sum the Sin series $x-x^3/3!+ x^5/5!-$ .... with an accuracy of 0.0001. Where x is the angle in radian.
49. To sum the Cosine series $1-x^2/2!+ x^4/4!-$ .... with an accuracy of 0.0001. Where x is the angle in radian.
50. To sum the series 1+1/2!+1/3!+.... up to n^{th} term.
51. To input a number and display the sum of its individual digits. Also display the number in reverse.
52. To input a number and to display whether it is a palindrome number.
53. To display the 5-digit palindrome numbers.
54. To display the palindrome numbers from 1 to 10000.

55. To display the palindrome numbers within a range.
56. To find the complete roots of a quadratic equation.
57. Generate the numbers between 1 and 'n', which is divisible by 'm', where n and m are integer numbers.
58. To input 10 numbers and to display their sum and average.
59. To input 'n' numbers and to display their sum and average.
60. To input 'n' numbers and to display the sum and average of nonnegative numbers.
61. To input 10 numbers and display the smallest and largest among them.
62. To input 'n' numbers and display the smallest and largest among them.
63. To count the number of numbers exactly divisible by 7 in a set of numbers.
64. To count the number of numbers exactly divisible by 7 in a set of numbers. Also display the smallest and largest among such numbers.
65. To display the number of numbers divisible by a given number in a set of numbers. Also display the smallest and largest among such numbers.
66. To check whether three sides a,b, and c can form a triangle. If so display its nature (Isosceles, Scalene or Equilateral) and area. The triangle cannot be formed if any of the side is negative or zero or if any length is grater than the sum of the other two.
67. To count the number of digits in an integer number.
68. To ask the user to type in a number. If the number is four digits long or longer, the program should give a message that the number is a large number. If it is two-digit long or shorter, then the computer should provide a message that the number is too small. Otherwise, print the message, 'Well done! We think alike!'
69. To input a four digit number and to display it in words.
70. To read the ages of few employees and count the number of persons in the age group of less than 25, 25-35, 36-45, 46-55 and greater than 55.
71. To prepare the electricity bill of customers for electricity board that charges the following rates to domestic users to discourage large consumption of energy.

    Up to 20 units: Re. 1 per unit.
    Up to 50 units: Rs. 1.50 per unit
    Up to 100 units: Rs. 2 per unit
    Up to 200 units: Rs. 2.50 per unit
    Up to 300 units: Rs. 3 per unit
    Above 300 units: Rs. 3.50 per unit

    If the total consumption exceeds 200 units, the consumer has to pay an additional surcharge of Rs. 2 per unit. Read the Consumer number

and the units consumed and display the amount due. The program should terminate only when a zero is entered to the consumer number.

72. To display the multiplication table 1×1=1 to 10×10 = 100.
73. To check whether a given year is leap year or not. The program should terminate only when a zero is entered.
74. To find the best matches for the equation 3x+2y-z=0 within the range of 1 to 10.
75. To display the count and percentage of occurrences of +ve, -ve and zeroes in a set of 'n' numbers.
76. To find whether a person is lucky, unlucky or average depending on the following. If sum of the individual digits of the year, in which the person is born, is divisible by three, the person is average lucky. If there exists a remainder of 1, then the person is lucky. The person is unlucky otherwise.
77. To display the first 'n' terms. First of which is 5. The terms are obtained by multiplying the preceding term by 3 successively.
78. To output the following pattern

```
*
* *
* * *
* * * *
* * * * *
```

up to 'n' lines

79. To output the following pattern

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

up to 'n' lines

80. To output the following pattern

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

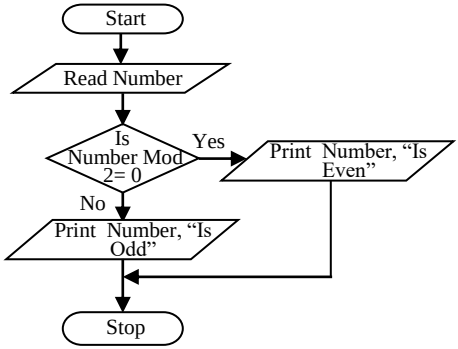
up to 'n' lines

81. To display the factorial of numbers from 1 to 20.
82. To display the factorial of odd numbers from 1 to 20.
83. To display the factorial of even numbers from 1 to 20.
84. To read a +ve integer less than 25 and to output the number, it's square and its cube in a tabular form.
85. To read a +ve integer less than 25 and to output each odd number, its square and its cube in a tabular form.

86. To read a +ve integer less than 25 and to output each even number, its square and its cube in a tabular format.
87. To classify the numbers within a rage into 'Perfect', 'Abundant' or 'Deficient' based on the following. A number is Perfect if the sum of its divisors (excluding the number itself) is equal to the number, e.g.,6. A number is Abundant if the sum of its divisors (excluding the number itself) is grater than the number, e.g., 12. A number is deficient if it is neither Perfect nor Abundant.
88. To generate a numerical calculator that can read two numbers and can perform basic arithmetic (+,-,× and ÷). Check that the second operand of ÷ operator cannot be zero.
89. To input the month number (1 to 12) and to display the name of the corresponding month.
90. To input the votes scored by 5 candidates. Their codes are 1, 2, 3, 4 and 5. The program should terminate when zero is entered and display the number and percentage of votes scored by each candidate. Also display the number and percentage of invalid votes.

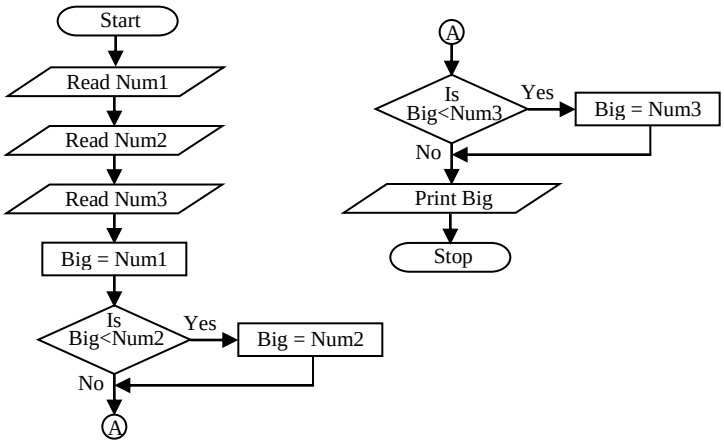
Answers to Select Questions

1.



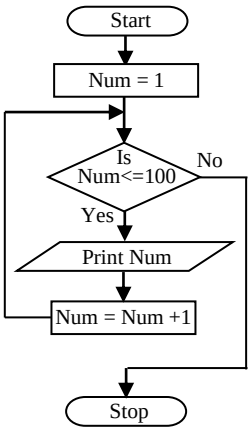

```
/* To display whether an integer is even or odd */
#include<stdio.h>
main()
{
      int number;
      clrscr();
      printf("Enter an integer number : ");
      scanf("%d",&number);
      if(number%2 == 0)
         printf("%d is an Even Number",number);
      else
         printf("%d is an Odd Number",number);
      getch();
}
```

3.



```
/* To display the largest of three numbers */
#include<stdio.h>
main()
{
      float num1,num2,num3,big;
      clrscr();
      printf("Enter Three Numbers\n");
      printf("First Number  : ",&num1);
      scanf("%f",&num1);
      printf("Second Number : ",&num2);
      scanf("%f",&num2);
      printf("Third Number  : ",&num3);
      scanf("%f",&num3);
      big = num1;
      if(big < num2)
            big=num2;
      if(big < num3)
            big=num3;
      printf("Largest among %g, %g and %g is %g", num1,num2,num3,big);
      getch();
}
```

5.

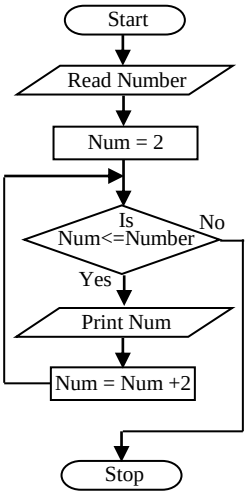



```
 /* To display the whole numbers from 1 to 100 */
#include<stdio.h>
main()
{
  int num;
  clrscr();
  printf("Whole number from 1 to 100 are : \n");
  for(num=1;num<=100;++num)
    printf("%d\t",num);
  getch();
}
```

12

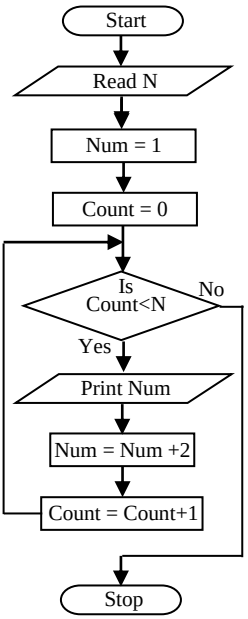
```
/* To display the even numbers up to a given number */
#include<stdio.h>
main()
{
```

```
   int number,num;
   clrscr();
   printf("Even numbers up to which number ? ");
   scanf("%d",&number);
   printf("The Even numbers from 2 to %d are : \n",
                                                  number);
   for(num=2;num<=number;num+=2)
     printf("%d\t",num);
   getch();
}
```



15.

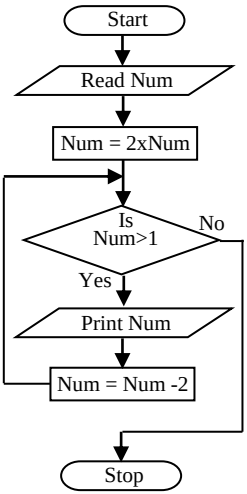
```
/* To display the first 'n' odd numbers */
#include<stdio.h>
main()
{
   int n,num=1,count;
   clrscr();
   printf("How many Odd numbers ? ");
   scanf("%d",&n);
    printf("The first %d Odd numbers are :\n",n);
   for(count=0;count<n;++count)
     {
       printf("%d\t",num);
       num+=2;
     }
   getch();
}
```



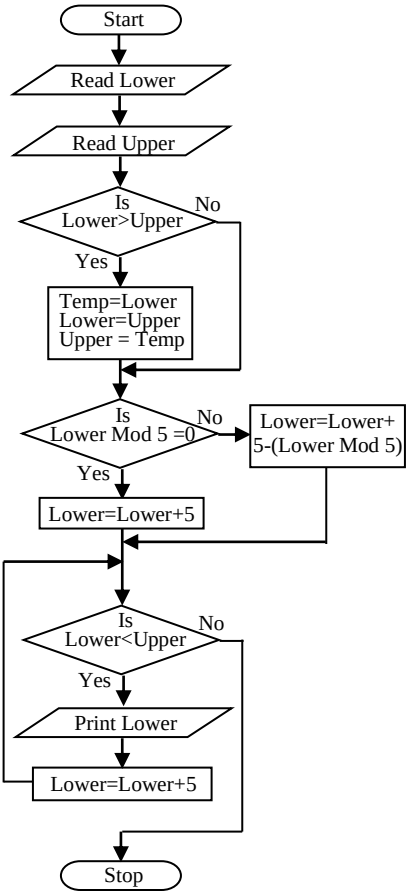
18.

```
/* To display the first 'n' even numbers in reverse order */
#include<stdio.h>
main()
```

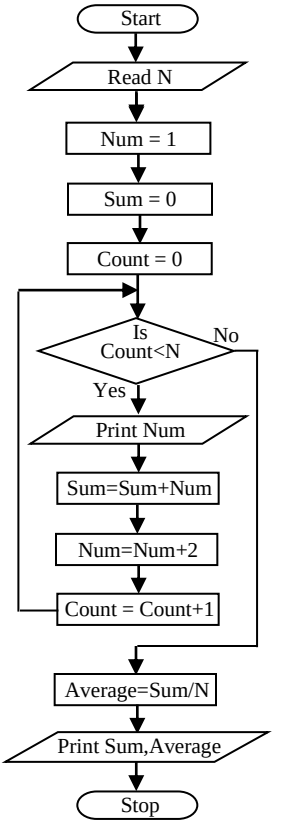
```
{
  int num;
  clrscr();
  printf("How many Even numbers ? ");
  scanf("%d",&num);
   printf("First %d even numbers in reverse
           order is : \n",num);
   num=num*2;
  for(;num>1;num-=2)
       printf("%d\t",num);
  getch();
}
```



22.



```
/* To display the multiples of 5 within a given range.
   Both the lower and upper bounds are excluded */
#include<stdio.h>
main()
{
      int lower,upper,num,temp;
      clrscr();
      printf("Enter the Lower bound : ");
      scanf("%d",&lower);
      printf("Enter the Upper bound : ");
      scanf("%d",&upper);
      if(lower>upper)
      {     temp = lower;
            lower = upper;
            upper = temp;
      }
      printf("The multiples of 5 between %d and %d are :\n", lower,upper);
      if (lower%5 == 0)
            lower = lower+5;
      else
            lower = lower+(5-(lower%5));
      for(;lower<upper;lower+=5)
                  printf("%d\t",lower);
      getch();
}
```

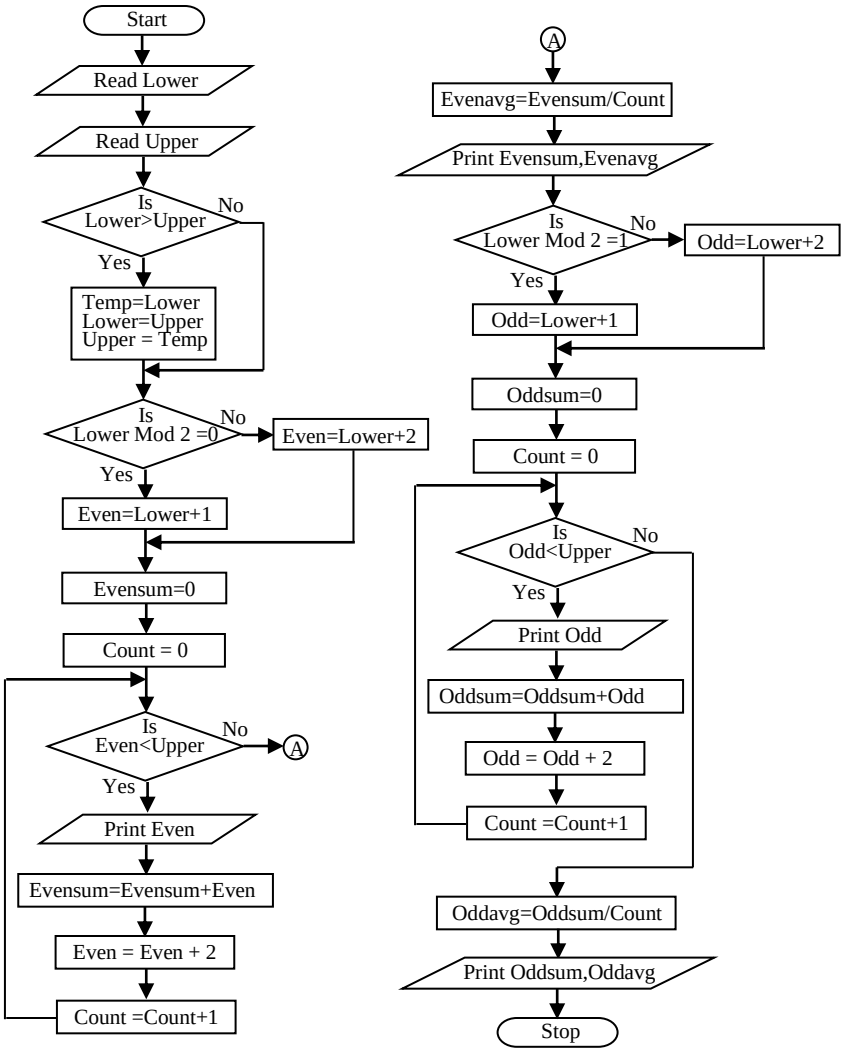


25.

```
/* To display the first 'n' odd numbers */
/* Also display their sum and average */
#include<stdio.h>
main()
{
   int n,num=1,count=0;
   float sum=0.0,average;
   clrscr();
   printf("How many Odd numbers ? ");
   scanf("%d",&n);
   printf("The first %d Odd numbers are :\n",n);
   for(;count<n;++count)
   {
            printf("%d\t",num);
            sum+=num;
            num+=2;
   }
```

```
    average = sum/n;
    printf("\nSum of the first %d Odd numbers is %g",n,sum);
    printf("\nAverage of the first %d Odd numbers is %g",n,average);
    getch();
}
```

29.

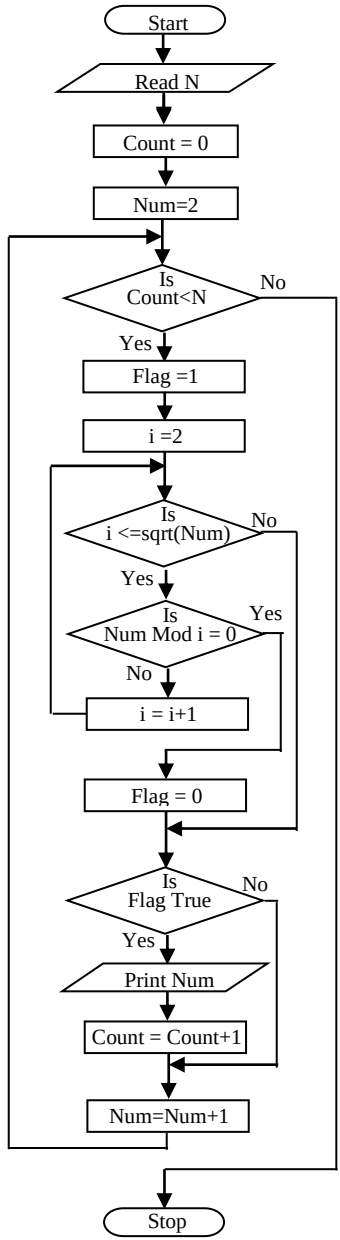


```
/* To display the Even and Odd numbers within a range
      Display the numbers separately. Also display their
      sum and average.
      Both the lower and upper bounds are exclusive */
#include<stdio.h>
main()
{
      int lower,upper,even,odd,count,temp;
      float evensum=.0,oddsum=0.0,evenavg,oddavg;
      clrscr();
      printf("Enter the Lower bound : ");
      scanf("%d",&lower);
      printf("Enter the Upper bound : ");
      scanf("%d",&upper);
      if(lower>upper)
      {     temp = lower;
            lower = upper;
            upper = temp;
      }
      if (lower%2 == 0)
            even = lower+2;
      else
            even = lower+1;
      printf("Even numbers between %d and %d are :\n",lower,upper);
      for(count=0;even<upper;even+=2,++count)
      {     printf("%d\t",even);
            evensum+=even;
      }
      evenavg=evensum/count;
      printf("\nSum of the even numbers from %d to %d is %g",
                                          lower,upper, evensum);
      printf("\nAverage of the even numbers from %d to %d is %g\n\n",
                                          lower,upper,evenavg);
      if (lower%2 == 1)
            odd = lower+2;
      else
            odd = lower+1;
      printf("Odd numbers between %d and %d are :\n",lower,upper);
      for(count=0;odd<upper;odd+=2,++count)
      {     printf("%d\t",odd);
            oddsum+=odd;
      }
      oddavg=oddsum/count;
```

```
        printf("\nSum of the odd numbers from %d to %d is %g",
                                                lower,upper,oddsum);
        printf("\nAverage of the odd numbers from %d to %d is %g",
                                                lower,upper,oddavg);
        getch();
}
```

32.

```
/* To display the first 'n' prime numbers */
#include<stdio.h>
#include<math.h>
main()
{
   int num,i,n,flag,count=0;
   clrscr();
   printf("How many prime numbers ? ");
   scanf("%d",&n);
   printf("First %d Prime Numbers are:\n",n);
   for(num=2;count<=n;++num)
   {  flag = 1;
      for(i= 2;i<=sqrt(num);++i)
         if( num%i == 0)
         {  flag=0;
            break;
         }
      if(flag)
      {  printf("%d\t",num);
         ++count;
      }
   }
   getch();
}
```



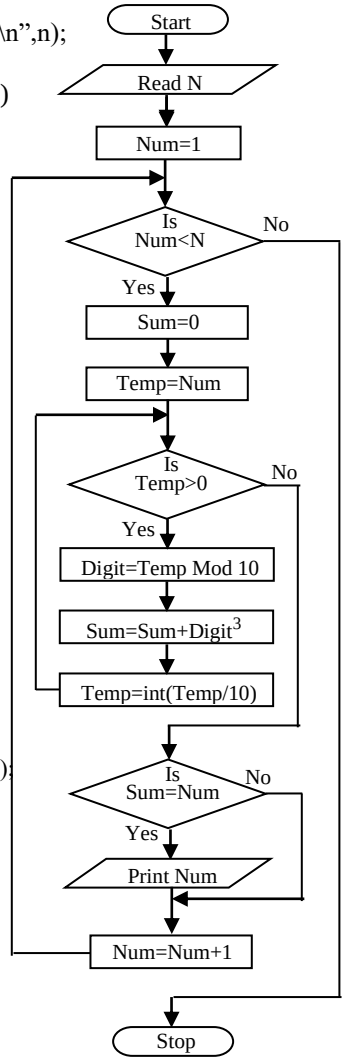
35.

```
/* To display the Armstrong numbers up to
    a given  number using for loops */
#include<stdio.h>
#include<math.h>
main()
{  int num,n,sum,digit,temp;
```

```
   clrscr();
   printf("Up to which number ? ");
   scanf("%d",&n);
   printf("Armstrong Numbers Up to %d are:\n",n);
   for(num=1;num<=n;++num)
   {  for(temp=num,sum=0;temp>0;temp/=10)
      {  digit=temp % 10;
         sum+=pow(digit,3);
      }
      if(num==sum)
         printf("%d\t",num);
   }
   getch();
}
```

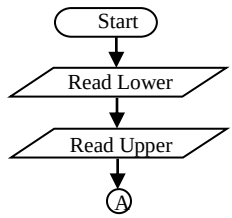
or

```
/* To display the Armstrong numbers up to a
    given  number using a for loop and
   a while loop */
#include<stdio.h>
#include<math.h>
main()
{  int num,n,sum,digit,temp;
   clrscr();
   printf("Up to which number ? ");
   scanf("%d",&n);
   printf("Armstrong Numbers Up to %d are:\n",n);
   for(num=1;num<=n;++num)
   {  temp=num;
      sum=0;
      while(temp>0)
      {  digit=temp % 10;
         sum+=pow(digit,3);
         temp/=10;
      }
      if(num==sum) printf("%d\t",num);
   }
   getch();
}
```

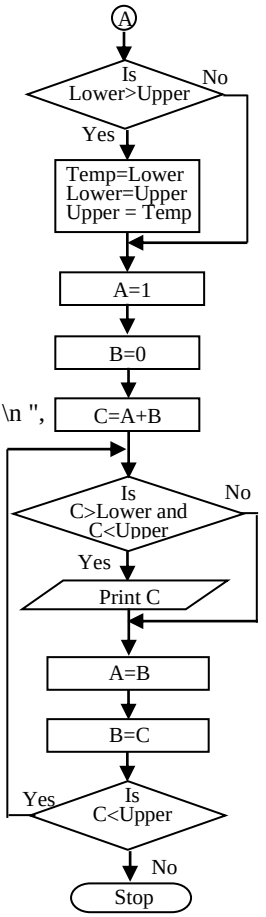


41.

```
/* To display the Fibonacci numbers within a range
```



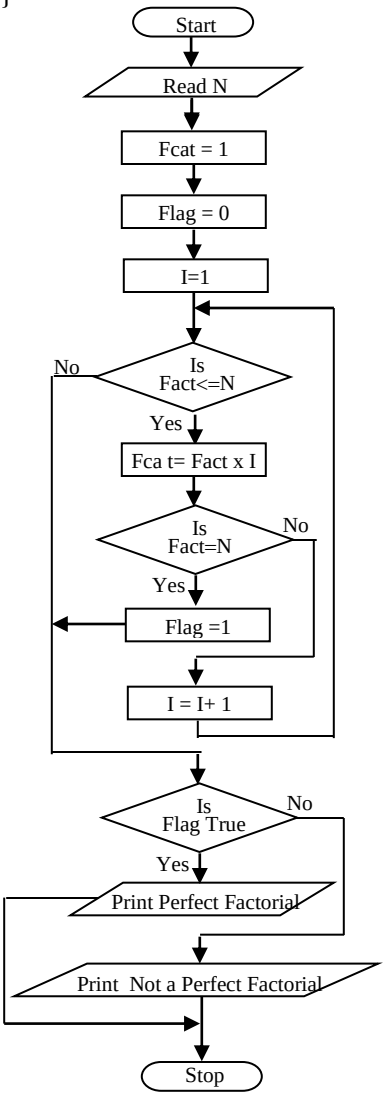
```
   Both upper and lower bounds are excluded */
#include<stdio.h>
main()
{  long lower,upper,a=1,b=0,c,temp;
   clrscr();
   printf("Enter a range\n");
   printf("What is the Lower bound ? ");
   scanf("%ld",&lower);
   printf("What is the upper bound ? ");
   scanf("%ld",&upper);
   if(lower>upper)
   {  temp=lower;
      lower=upper;
      upper=temp;
   }
   printf("Fibonacci numbers between %ld and %ld are:\n ",
                                          lower,upper);
   do
   {  c=a+b;
      a=b;
      b=c;
      if((c>lower) && (c<upper))
         printf("%ld\t",c);
   }
   while (c<upper);
   getch();
}
```



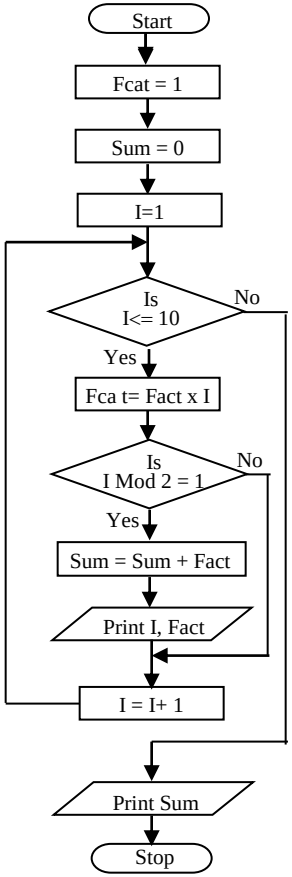
43.

```
/* To check whether a given number is
   perfect factorial or not */
#include<stdio.h>
main()
{
   int n,i=1,flag=0;
   long fact=1;
   clrscr();
   printf("Enter your number : ");
   scanf("%d",&n);
   for(;fact<=n;++i,fact*=i)
      if(fact==n)
         { flag=1;
            break;
```

```
        }
    if(flag)
      printf("%d is a perfect factorial",n);
    else
      printf("%d is not a perfect factorial",n);
    getch();
}
```



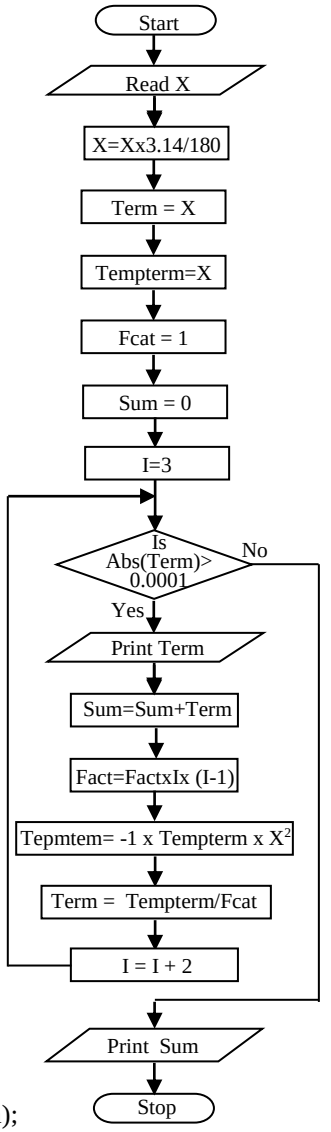
Flowchart of question no.43



Flowchart of question no.45

45.

```
/* To display the factorials of odd numbers from 1 to 10 and their sum */
#include<stdio.h>
main()
{
   int i=1;
   float fact=1.0,sum=0.0;
   clrscr();
   printf("Odd Number\tFactorial\n");
   for(;i<=10;++i,fact*=i)
      if((i%2)==1)
         {  printf("   %d\t\t  %g\n",i,fact);
            sum+=fact;
         }
   printf("Sum of above factorials is %g",sum);
   getch();
}
```



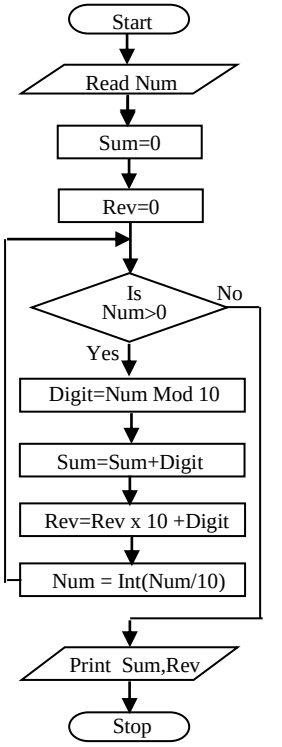
48.

```
/* To Sum the sine series */
#include<stdio.h>
#include<math.h>
main()
{
   int i=3;
   float x,tempx;
   double fact=1.0,sum=0.0,term,tempterm;
   clrscr();
   printf("Enter the angle in degree : ");
   scanf("%f",&x);
   tempx=x;
   x=x*3.14/180.0;
   term=tempterm=x;
   printf("The terms are:\n");
   for(;fabs(term)>0.0001;i+=2)
   {  printf("%g\t",term);
     sum+=term;
      fact=fact*i*(i-1);
      tempterm=(-1*tempterm*x*x);
      term=tempterm/fact;
   }
   printf("\n\nSin(%g) = %.2f %.2f",tempx,sum);
```

```
   getch();
}
```

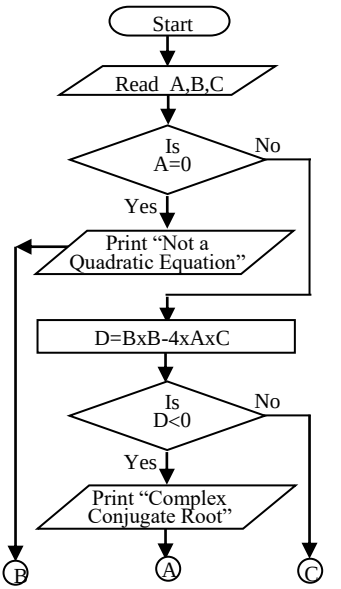
51.

```
/* To Sum the individual digits of a number
   Also display the number in reverse */
#include<stdio.h>
main()
{
   int sum=0,digit;
   long num,temp,rev=0;
   clrscr();
   printf("Enter a Number : ");
   scanf("%ld",&num);
   temp=num;
   while(num>0)
   {  digit=num%10;
      sum+=digit;
      rev=rev*10+digit;
      num=(int)num/10;
   }
   printf("Sum of the digits of %ld is %d\n",temp,sum);
   printf("Reverse of %ld is %ld\n",temp,rev);
   getch();
}
```

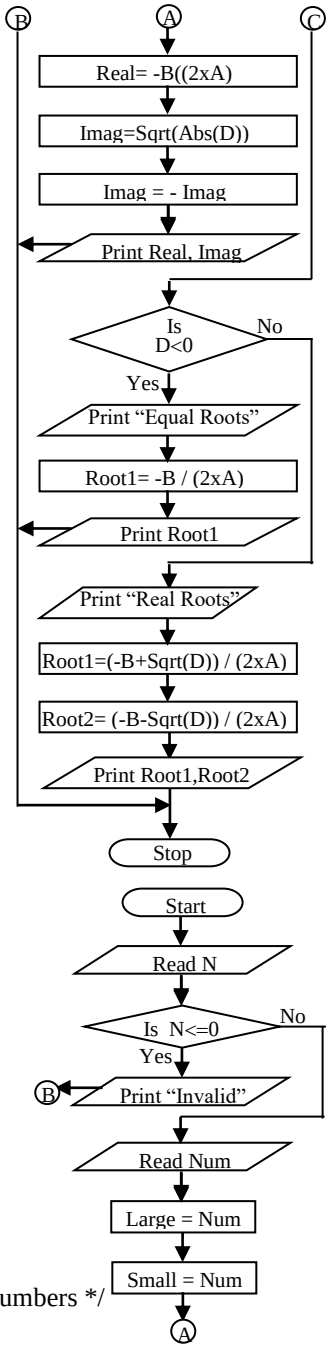


55.

```
/* To Find the roots of a quadratic equation */
#include<stdio.h>
#include<math.h>
main()
{  float a,b,c,d,root1,root2,imag,real;
   clrscr();
   printf("Enter the coefficients\n");
   printf(" Value of a ? ");
   scanf("%f",&a);
   printf(" Value of b ? ");
   scanf("%f",&b);
   printf(" Value of c ? ");
   scanf("%f",&c);
```



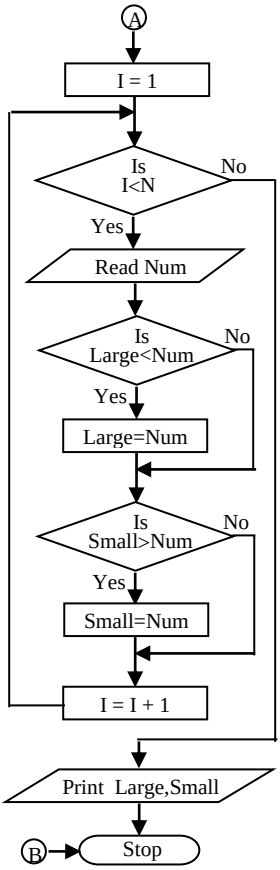
```
   if (a==0)
      printf("This is not a Quadratic Equation");
   else
   {  d=(b*b)-(4*a*c);
      if(d<0)
      {  printf("Complex conjugate roots\n");
         real= -b/(2.0*a);
         imag= sqrt(fabs(d))/(2.0*a);
         imag=-imag;
         printf("Real part is %g\n",real);
         printf("Imaginary part is %g\n",imag);
      }
      else
         if(d==0)
         {  printf("Equal roots\n");
            root1= -b/(2.0*a);
            printf("Root is %g\n",root1);
         }
         else
         {  printf("Real roots\n");
            root1= (-b+sqrt(d))/(2.0*a);
            root2= (-b-(sqrt(d)))/(2.0*a);
            printf("Root - 1 is %g\n",root1);
            printf("Root - 2 is %g\n",root2);
         }
      }
   getch();
}
```



61.

```
/* To Find the largest and smallest among 'n' numbers */
#include<stdio.h>
main()
```

```
{  int i=1,n;
   float num,large,small;
   clrscr();
   printf("How many number ? ");
   scanf("%d",&n);
   if(n<=0)
      printf("Invalid Input");
   else
   {
      printf("Enter number - %d : ",i);
      scanf("%f",&num);
      large=small=num;
      for(;i<n;++i)
      {  printf("Enter number - %d : ",i+1);
         scanf("%f",&num);
         if (large<num)
            large = num;
         if(small>num)
            small = num;
      }
      printf("Largest  is %g\n",large);
      printf("Smallest is %g",small);
   }
   getch();
}
```



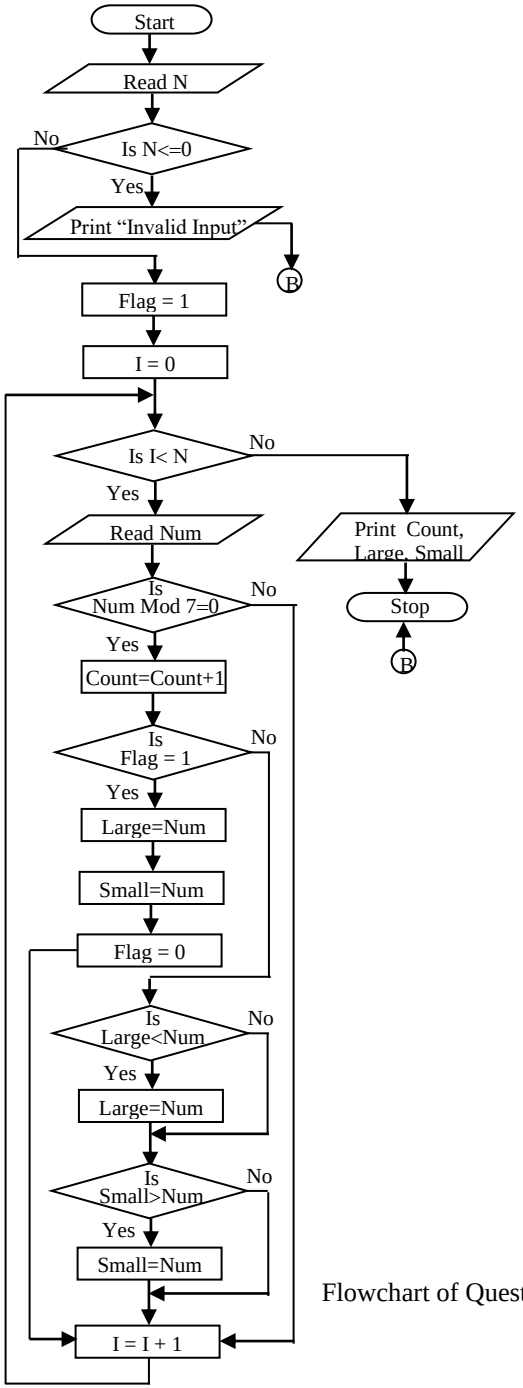
63.

```
/* To count the number of numbers divisible by 7
   in a given set of numbers. Also find the largest
   and smallest among such numbers */
#include<stdio.h>
main()
{  int i=0,flag=1,count=0,n,num,large,small;
   clrscr();
   printf("How many number ? ");
   scanf("%d",&n);
   if(n<=0)
      printf("Invalid Input");
   else
   {  for(;i<n;++i)
      {  printf("Enter number - %d : ",i+1);
         scanf("%d",&num);
         if (num%7==0)
```

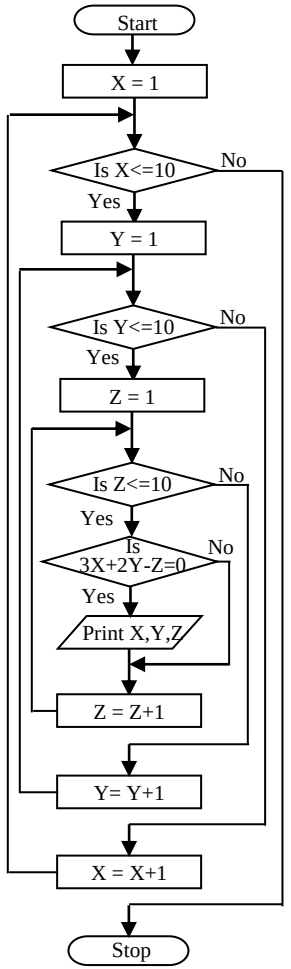
```
        {  count++;
           if(flag)
           {  large=small=num;
              flag=0;
           }
           else
           {  if(large<num)
                 large = num;
              if(small>num)
                 small = num;
           }
        }
     }
     if(!flag)
     {  printf("There are %d number divisible by 7\n",count);
        printf("Largest among such numbers is %d\n",large);
        printf("Smallest among such numbers is %d",small);
     }
     else
     printf("There exist no numbers divisible by 7");
  }
  getch();
}
```

73.

```
/* To find the best matches for the expression
   3x+2y-z=0 within the range x,y,z=1 to 10 */
#include<stdio.h>
main()
{  int x,y,z;
   clrscr();
   printf("The best matches are :\n");
   for(x=1;x<=10;++x)
      for(y=1;y<=10;++y)
         for(z=1;z<=10;++z)
            if(3*x+2*y-z==0)
               printf("x = %d\ty = %d\tz = %d\n",x,y,z);
   getch();
}
```



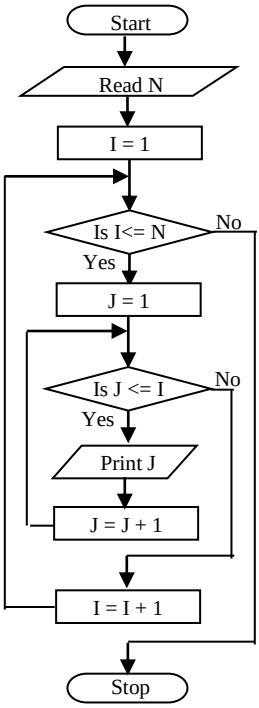
Flowchart of Question 63



Flowchart of Question 73

78.

```
/* To generate the pattern */
#include<stdio.h>
#include<conio.h>
main()
{  int i=1,j=1,n,x=3,y=40;
   clrscr();
   printf("How many lines ? ");
   scanf("%d",&n);
   gotoxy(y,x);
   for(;i<=n;++i,++x,y-=2)
   {  gotoxy(y,x);
      for(j=1;j<=i;++j)
         printf("%4d",j);
      printf("\n");
   }
   getch();
}
```



Flowchart for Question 78

UNIT V

FUNCTIONS

5.1. Introduction

A function is a self-contained program segment that carries out some specific, well-defined task. A function will carry out its intended action whenever it is accessed (i.e., whenever the function is called). The same function can be accessed from several different places within a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed. C supports the use of two types of functions

- Library functions, which are used to carry out a number of commonly used operations or calculations, and
- Programmer defined functions, which are defined by the programmer for carrying out various individual tasks.

The use of programmer-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which can individually be solved. This is known as modularisation. The advantages of modularisation are:

- Avoids the need for redundant coding
- If a particular group of instructions to be accessed repeatedly, from several different places within a program, it can be placed within a function, which can then be accessed whenever it is needed. Thus the use function avoids the need for redundant programming of the same instructions.
- Increase the logical clarity
- Decomposition of lengthy, complicated program into several concise functions will improve the logical clarity of the program. Such programs are easier to write, read and debug.
- Enables the 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. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process.

5.2. Defining a Function

A function definition has two principal components: the function header and the body of the function. The function header contains the type specification of the value returned by the function, followed by the function name, and (optionally) 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 terms, the function header statement can be written as

data-type name(type1 arg1,type2 arg2,...,typen argn)

where data-type represents the data type of the item that is returned by the function, name represents the function name, and type1, type2, ..., typen represents the data types of the arguments arg1, arg2, ..., argn. The data types are assumed to be of type integer if they are not shown explicitly.

The arguments are called formal arguments or formal parameters, because they represent the names of the data item that are transferred into the function from the calling portion of the program. The corresponding arguments in the function reference or function call are called actual arguments or actual parameters, since they define the data items that are actually transferred.

The body of the function is a compound statement that defines the action to be taken by the function. This compound statement can contain expression statements, other compound statements, control statements, and so on.

The body of the function should include one or more return statements, in order to return a value to the calling portion of the program. Information is returned from the function to the calling portion of the program via the return statement. The return statement also causes the program logic to return to the point from which the function was accessed.
In general terms the return statement is written as

return expression;

The value of the expression is returned to the calling portion of the program.

For example, the following C program determines the largest of three integer quantities. This program makes use of the function that determines the larger of two integer quantities.

```
/* The main function */
#include <stdio.h>
int max(int a, int b);
main()
{  int x,y,z;
   clrscr();
   printf("First No  : ");
   scanf("%d",&x);
   printf("Second No : ");
   scanf("%d",&y);
   printf("Third No  : ");
   scanf("%d",&z);
```

```
   printf("Largest among %d , %d and %d is %d" ,
                        x,y,z,max(max(x,y),z));
   getch();
}

/* Function that returns the larger of two integer quantities*/
int max(int a, int b)          /* Function header */
{  int k;                      /* Body of the function */
   k=(a>b)?a:b;
   return(k);
}                              /* End of the function */
```

The first line of the function contains the function name, max, followed by the formal augments a and b, enclosed in parentheses. The formal arguments a and b represents the data items that are transferred to the function from the calling portion of the program. The function name max is preceded by the data type int, which describes the data item that is returned by the function. In addition, the formal arguments a and b are preceded by the data type int. The body of the function begins on the second line. Within the body of the function, a local variable k is defined. Following the declaration of k is a conditional expression statement that assigns the larger of a and b. Finally, the return statement causes the larger value to be returned to the calling portion of the program.

The expression is optional within a return statement. If expression is omitted, the return statement simply causes control to revert back to the calling portion of the program, without any transfer of information. The return statement can be absent altogether from a function definition, though this is generally regarded as poor programming practice. If a function reaches the end without encountering a return statement, control simply reverts back to the calling portion of the program without returning any information.

For example, the following program illustrates the use of a function that accepts two integer quantities and determines the larger value, which is then displayed. The function does not return any information to the calling program.

```
/*Illustrating the use of return statement without expression*/
#include <stdio.h>
void max(int a, int b);
main()
{  int x,y;
   clrscr();
   printf("First No  : ");
   scanf("%d",&x);
```

```
   printf("Second No : ");
   scanf("%d",&y);
   max(x,y);
   getch();
}

/* Function without an expression in return statement */
void max(int a, int b)
{  int k;
   k=(a>b)?a:b;
   printf("Largest among %d and %d is %d",a,b,k);
   return;
}
```

Only one expression can be included in the return statement. Thus, a function can return only one value to the calling portion of the program via return statement. A function definition can include multiple return statements, each containing a different expression. At any time, only one return statement will be executed.

```
/* Illustrating the use of two return statements */
#include <stdio.h>
int max(int a, int b);
main()
{  int x,y,z;
   clrscr();
   printf("First No  : ");
   scanf("%d",&x);
   printf("Second No : ");
   scanf("%d",&y);
   printf("Third No  : ");
   scanf("%d",&z);
   printf("Largest among %d , %d and %d is %d",
   x,y,z,max(max(x,y),z));
   getch();
}

/* Function contains two return statements */
int max(int a, int b)
{  if (a>b)
      return(a);
   else
      return(b);
}
```

The key word void can be used as a type specifier when defining a function that does not return anything or when function definition does not include any arguments.

```
/* Illustrating the use of void as type specifier */
#include <stdio.h>
void max(void);
int x,y;  /* Global variables */
main()
{  clrscr();
   printf("First No  : ");
   scanf("%d",&x);
   printf("Second No : ");
   scanf("%d",&y);
   max();
   getch();
}

void max(void)
{  int k;
   k=(x>y)?x:y;
   printf("Largest among %d and %d is %d",x,y,k);
   return;
}
```

5.3. Accessing a Function

A function can be accessed (i.e., called) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function. The function call may be a part of simple expression, or it may be one of the operands within a more complex expression.

The arguments appearing in the function call are referred to as actual parameters and the arguments appearing in the header line of the function definition is known as formal parameters. In a normal function call, there will be one actual argument for each formal argument. The actual arguments may be expressed as constants, single variables, or more complex expressions. Each actual argument must be of the same data type as its corresponding formal arguments.

For example, consider the following C program which reads in two integer variable and displays the larger of the two.

```
/* The main function */
#include <stdio.h>
int max(int a, int b);
main()
{  int x,y;
   clrscr();
   printf(“First No  : ”);
   scanf(“%d”,&x);
   printf(“Second No : ”);
   scanf(“%d”,&y);
   printf(“Largest of  %d and %d is %d” , x,y,max(x,y));
   getch();
}

/* Function that returns the larger of two integer quantities*/
int max(int a, int b)
{  int k;
   k=(a>b)?a:b;
   return(k);
}
```

Within the program, main contain only one call to the programmer defined function max. The call is the part of the printf function. The function call contains two actual arguments, the integer type variables x and y. When the function is accessed, the value of x and y are transferred to the function. These values are represented by a and b within the function. The larger of two data is determined and returned to the calling portion of the program, where it is displayed on the standard output device.

There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. For example, the following program illustrates the repeated use of the function max to determine the largest among a set of numbers.

```
/* The main function */
#include <stdio.h>
float max(float a, float b);
main()
{  float num,large;
   int n,i=1;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
    printf("Enter number - %d : ",i);
   scanf("%f",&num);
```

```
    large = num;
    for(;i<n;++i)
    {  printf("Enter number - %d : ",i+1);
       scanf("%f",&num);
       large = max(large,num);
    }
    printf("Largest is %g",large);
    getch();
}

/* Function that returns the larger of two integer quantities*/
float max(float a, float b)
{  float k;
   k=(a>b)?a:b;
   return(k);
}
```

The function max will be accessed ‘n’ times within the for loop.

5.4. Function Prototypes

If the function call precedes the function definition, the function must be declared within the calling function. A function prototype is used for this purpose. Function prototypes are usually written at the beginning of the program following the include statement. A function prototype specifies the type of value returned by the function, the name of the function and the type(s) of argument(s) expected by the function. This enables the compiler to check that, when the function is used, the correct number and type of arguments are supplied. The general form of the function prototype is

data-type name(type1 arg1,type2 arg2, ...,typen argn);

where data-type represents the data type of the item that is returned by the function, name represents the function name, and type1, type2, ..., typen represents the data types of the arguments arg1, arg2, ..., argn. Thus, the function prototype resembles the header statement of a function definition, except that the prototype is terminated by a semicolon.

For example, consider the following program that calculates the factorial of a positive integer quantity. The program utilizes the function named factorial to determine the factorial of a positive integer. The reference to the function factorial within the main function appears ahead of ifs definition. Therefore a function prototype is included in the beginning of the program following the include statement. The function prototype indicates that a function called factorial, which accepts an integer quantity and returns a double precision quantity, will be defined later in the program.

```
/* The main function */
#include <stdio.h>
double fact(int x);      /* Function prototype */
main()
{  int n;
   clrscr();
   printf(" Enter a positive number : ");
   scanf("%d",&n);
   if(n<0)
      printf("Invalid Input");
   else
      printf("factorial of %d is %g",n,fact(n));
   getch();
}

/* Function that calculates and return the factorial of an integer */
double fact(int x)
{  int i;
   double f = 1.0;
      for(i=2;i<=x;++i)
         f*=i;
   return f;
}
```

5.5. Passing Arguments to a Function

In C language, arguments can be passed to a function in two different ways:

- Argument passing by value and
- Argument passing by reference

When argument passing by value technique is employed, the value of the actual parameters is copied into the corresponding formal parameters. Any changes made to the formal parameters within the called functions will not be reflected in the value of actual parameter within the calling function. Passing an argument by value has it own advantages and disadvantages. The advantage is that the passing by value technique allows a single valued actual argument to be written as an expression rather than being restricted to a single variable. Another advantage of passing arguments by value is that it protects the value of this variable from alterations within the calling function. On the other hand, it does not allow information to be transferred

back to the calling portion of the program via arguments. Thus, passing by value is restricted to a one-way transfer of information.

For example, the following program illustrates the side effects of passing arguments by value mechanism.

```
/* Illustrating the effect of argument passing by value */
#include <stdio.h>
void modify(int x, int y);
main()
{  int x=1,y=2;
   clrscr();
   printf("Before modifying within main()\n");
   printf("\t\tx = %d and y= %d\n",x,y);
   x++;
   ++y;
   printf("After modifying within main()\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   modify(x,y);
   printf("After returning from the modify function to the main()\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   getch();
}

void modify(int x, int y)
   {  printf("Before modifying within the modify function\n");
      printf("\t\tx = %d and y = %d\n",x,y);
      x++;
      ++y;
      printf("After modifying within function :\n");
      printf("\t\tx = %d and y = %d\n",x,y);
      return;
   }
```

This program generates the following output.

```
Before modifying within main()
        x = 1 and y= 2
After modifying within main()
        x = 2 and y = 3
Before modifying within the modify function
        x = 2 and y = 3
After modifying within function:
        x = 3 and y = 4
After returning from the modify function to the main()
        x = 2 and y = 3
```

This example illustrates that when arguments are passed by value, the values of the actual arguments are copied into their corresponding formal arguments. The values of the formal arguments can be altered within the function, but the values of their corresponding actual arguments within the calling function will not change.

When passing by reference technique is used, the address of the data item (actual parameter) is passed to the called function. This allows the data item within the calling portion of the program to be accessed by the called function, altered within the called function, and then returned to the calling portion of the program in altered form. The contents of the address can be accessed freely, either within the called function or within the calling function. Any change that is made to the data item will be recognized in both the called function and the calling function. Arguments such as array name and pointers use this type of argument transfer. For example, if an array name is specified as an actual argument, the individual array elements are not copied. Instead, the location of the array (i.e., the location of the first element) is passed to the function. If an element of an array is then accessed within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, an alteration to an array element within the function will carry over to the calling routine.

5.6. Storage Classes

Data type and storage class are the two different ways to characterize variables. Data type refers to the type of information represented by a variable, e.g., integer, floating-point, character, etc. Storage class refers to the permanence of a variable, and its scope within the program, i.e., the portion of the program over which the variable is recognized.

There are four storage class specifications in C: Automatic, External, Static and Register. They are identified by the keywords auto, extern, static and register, respectively. The storage class associated with a variable can sometimes be established simply by the location of the variable declaration within the program. In other situations, however, the keyword that specifies a particular storage class must be placed at the beginning of the variable declaration.

Automatic (Local) Variables

Automatic variable are always declared within a function and are local to the function in which they are declared, i.e., their scope is confined to that function. Automatic variables defined in different functions will

therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as automatic variable unless a different storage class specification is shown within the declaration. This includes formal argument declarations. A variable can be explicitly defined as automatic variable by placing the keyword auto at the beginning of the variable declaration. Since the location of the variable declarations within the program determines the automatic storage class, the keyword auto is not required at the beginning of each variable declaration.

For example, consider a program for checking whether a given number is prime or not. Within main(), num is an automatic variable. Within the function prime, flag and i are automatic variables.

```
#include<stdio.h>
#include<math.h>
int prime(int x);
main()
{  auto int num;
   clrscr();
   printf("Enter a Number : ");
   scanf("%d",&num);
   if(num<0)
      printf(“Invalid Input”);
   else
      if(prime(num))
         printf("%d is a Prime number");
      else
         printf("%d is not a Prime number");
   getch();
}

int prime(int x)
{  auto int flag = 0,i;
   for(i=2;i<=sqrt(x);++i)
      if(x%i==0)
         return flag;
   flag = 1;
   return flag;
}
```

Automatic variables can be assigned initial values by including appropriate expressions within the variable declaration or by explicit assignment expressions elsewhere in the function. Such values are assigned each time the function is re-entered. If an automatic variable is not initialised in some manner, its initial value will be unpredictable. An automatic variable

does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited.

To illustrate these points, consider the following program that displays the factorial of numbers from 1 to 25. Note that both the functions main and fact contain one automatic variable named i. The automatic variable i has different meanings within each function and are independent of one another.

```
/* To display the factorial of number from 1 to 25*/
#include <stdio.h>
double fact(int n);
main()
{  auto int i;
   clrscr();
   printf("Number\t\tFactorial\n");
   for(i=1;i<=25;++i)
      printf("%d\t\t%g\n",i,fact(i));
   getch();
}

/* Function that calculates and return the factorial of an integer */
double fact(int n)
{  auto int i;
   double f = 1.0;
      for(i=2;i<=n;++i)
         f*=i;
   return f;
}
```

External (Global) Variables

External variables are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. Since the external variables are recognized globally, they can be accessed from any function that falls within their scope. They retain their assigned values within this scope. Therefore, an external variable can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.

The use of external variables provides a convenient mechanism for transferring information back and forth between functions. In particular, we can transfer information into a function without using arguments. This is convenient when a function requires numerous input data items. We can also

transfer multiple data items out of a function (return statement can return only one data item).

When working with external variables, we must distinguish between external variable definition and external variable declaration. An external variable definition is written the same manner as an ordinary variable declaration. It must appear outside of, and usually before, the functions that accesses the external variables. An external variable definition will automatically allocate the required storage space for the external variable within the computer's memory. External variables can be assigned initial values as a part of variable definitions. If the initial value is not included in the definition of an external variable, the variable will be automatically assigned a value of zero. The storage-class specifier extern is not required in an external variable definition, since the external variables will be identified by the location of their definition within the program.

If a function requires an external variable that has been defined earlier in the program, then the function may access the external variable freely, without any special declaration within the program. On the other hand, if the function definition precedes the external variable definition, then the function must include a declaration for that external variable. An external function declaration must begin with the storage-class specifier extern. The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside the function. Storage space will not be allocated to external variables as a result of external variable declaration. Moreover, an external variable declaration cannot include the assignment of initial values. The following sample program illustrates the definition, declaration and use of global variables. The output of this sample program illustrates that the any alteration to the value of an external variable within a function will be recognized within the entire scope of the external variable.

```
/* Illustrating the use of global variables  */
#include <stdio.h>
void modify(void)
{ extern int x,y; /* Global variable declaration */
   printf("Before modifying within the modify function\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   x++;
   ++y;
   printf("After modifying within function :\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   return;
}

int x=1,y=2; /* Global variable definition */
```

```
void modify(void); /* Function prototype */
main()
{ clrscr();
   printf("Before modifying within main()\n");
   printf("\t\tx = %d and y= %d\n",x,y);
   x++;
   ++y;
   printf("After modifying within main()\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   modify();
   printf("After returning from the modify function to the main()\n");
   printf("\t\tx = %d and y = %d\n",x,y);
   getch();
}
```

This sample program will generate the following output.

```
Before modifying within main()
        x = 1 and y= 2
After modifying within main()
        x = 2 and y = 3
Before modifying within the modify function
        x = 2 and y = 3
After modifying within function :
        x = 3 and y = 4
After returning from the modify function to the main()
        x = 3 and y = 4
```

Static Variables

A variable is declared to be static by prefixing its normal declaration with the keyword static, as in

```
static int fibonacci;
```

We could also use

```
int static fibonacci;
```

since the properties of a variable may be stated in any order. Initial values can be included in the static variable declaration as in

```
int static fibonacci = 1;
```

In the absence of an explicit initialisation, C guarantees that static variables will be initialised to zero. A static variable can be either internal or external. The declaration of an internal static variable appears inside a function, as in

```
fun()
{
   ...
   static int fibonacci = 1;
```

```
    ...
}
```

An internal static variable have the same scope as automatic variables, i.e., they are local to the functions in which they are defined. Unlike automatic variables, however, the static variables retain their values throughout the life of the program. Thus, if a function is exited and then re-entered at a later time, the static variables defined within the function will retain their former values.

This is illustrated in following program that calculates the successive Fibonacci numbers. Note that f1 and f2 are static variables that are each assigned initial values. These initial values are assigned only once, at the beginning of the program execution. The subsequent values are retained between successive function calls, as they are assigned.

```
#include<stdio.h>
double fib(int n);
main()
{  int n,i;
   clrscr();
   printf("How many Fibonacci Number ? ");
   scanf("%d",&n);
   printf("The first %d Fibonacci Numbers are :\n",n);
   for(i=1;i<=n;++i)
      printf("%g\t",fib(i));
   getch();
}

double fib(int n)
{  static double f1=1.0,f2=0.0;
   double f;
   f=f1+f2;
   f1=f2;
   f2=f;
   return f;
}
```

The declaration of an external static variable appears outside of any function, as in

```
static int fibonacci = 1;
fun()
{
   ...
   ...
}
```

then the variable is known in the remainder of the file containing the declaration. The difference between the external variable and the external static variable is that the latter is unknown outside of the file in which it is declared. Thus, the scope of external static variable is limited to one file only.

The word static denotes permanence. Whether internal or external, a static variable is allocated storage only once, which is retained for the duration of the program. A static variable also affords a degree of 'privacy'. If it is internal, it is known only in the function in which it is declared. If it is external, it is known only in the file in which it is declared.

Register Variables

Registers are the special storage areas within the computer's central processing unit. The actual arithmetic and logical operations that comprise a program are carried out within these registers. Normally, these operations are carried out by transferring information from the computer's memory to these registers, carrying out the indicated operations, and then transferring the results back to the computer's memory. This general procedure is repeated many times during the course of a program's execution.

The execution time of a program can be reduced considerably if certain values can be stored within the these registers rather than in computer's memory. Such programs may also be somewhat smaller in size (i.e., they may require fewer instructions), since fewer data transfers will be required.

In C, the values of the register variables are stored within the registers of the central processing unit. A variable can be assigned this storage class simply by preceding the type declaration with the keyword register. This storage class is usually applied to a variable which will be heavily used in the program. In any case, only int, char and pointer variables may be declared as register variables. Furthermore, register may be applied only to automatic variables and to the formal parameters of functions.

The register and automatic storage classes are closely related. In particular, their scope is the same. Thus, the register variables, like automatic variables, are local to the function in which they are declared. Furthermore, the rules governing the use of register variables are the same as those for automatic variables, except that the address operator (&) cannot be applied to the register variables. Where possible, a register variable is assigned to a CPU register, rather than a normal memory location. If there are more register declarations than available CPU registers, then the word register is ignored for the excess declarations. If a register declaration is not honoured, the variables will be treated as having the storage class automatic. Unfortunately, there is no way determine whether a register declaration will

be honoured, other than to run a program with and without declaration and compare the result. A program that makes use of register variables should run faster than the corresponding program without register variables.

5.7. Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in recursive form, and second, the problem statement must include a stopping condition

As an example, consider the problem of finding the factorial of a positive integer quantity.

$$n! = 1 \times 2 \times 2 \times ... \times n,$$

where n is a positive integer.
This can be expressed as

$$n! = n \times (n-1)!$$

This is a recursive statement of the problem, in which the desired action n! is expressed in terms of a previous result, i.e., (n-1)!, which is assumed to be known. Also, we know that 0! = 1. This provides a stopping condition for the recursion.

```
/* Factorial using recursion */
#include<stdio.h>
double fact(int n);
main()
{
   int num;
   clrscr();
   printf("Enter a Number : ");
   scanf("%d",&num);
   if(num<0)
      printf("Invalid Input");
   else
      printf("%d! is %g ",num,fact(num));
   getch();
}

double fact(int n)
{
```

```
    if(n==0)
    return 1;
    else
    return n*fact(n-1)
}
```

When the program is executed, the function fact is accessed repeatedly, once in main and (n-1) times within itself. When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a stack until the condition that terminates the recursion is encountered. (A stack is last-in first-out data structure in which successive data items are 'pushed down' upon preceding data items. The items are later removed (i.e., they are popped) from the stack in reverse order). The function calls are then executed in reverse order. Thus, while evaluating a factorial recursively, the function call will proceed in the following order.

$n! = n \times (n-1)!$
$(n-1)! = (n-1) \times (n-2)!$
$(n-2)! = (n-2) \times (n-3)!$
.....
$2! = 2 \times 1!$
$1! = 1 \times 0!$

The actual values will then be returned in the following reverse order.

$0! = 1$
$1! = 1 \times 0! = 1 \times 1 = 1$
$2! = 2 \times 1! = 2 \times 1 = 2$
$3! = 3 \times 2! = 3 \times 2 = 6$
$4! = 4 \times 3! = 4 \times 6 = 24$
.....
$n! = n \times (n-1)! =$ .....

If a recursive function contains local variables, a different set of local variables will be created during each call. The names of the local variables will be the same, as declared within the function. However, the variables will represent a different set of values each time the function is executed. Each set of values will be stored on the stack, so that they will be available as various function calls are popped off the stack and executed.

The recursive program for converting a decimal number to its corresponding binary equivalent is illustrated below. To print the binary equivalent of n, print the binary equivalent of n/2 (integer division) and then prints n%2 (remainder).

```
#include<stdio.h>
void binary(long n);
main()
```

```
{  long num;
   clrscr();
   printf("Enter a Decimal Number : ");
   scanf("%ld",&num);
   printf("Binary equivalent of the Decimal Number %ld is ");
   binary(num);
   getch();
}

void binary(long n)
{  if(n>0)
   {  binary(n/2);
      printf("%d",n%2);
   }
}
```

Here is another example of recursive program that calculates the successive Fibonacci numbers.

```
double fib(int n);
#include<stdio.h>
main()
{  long n,i;
   clrscr();
   printf("How many Fibonacci numbers ? ");
   scanf("%ld",&n);
   printf("The first %d Fibonacci Numbers are :\n");
   for(i=1;i<=n;++i)
      printf("%g\t",fib(i));
   getch();
}

double fib(int n)
{  if(n<3)
      return 1;
   else
      return fib(n-1)+fib(n-2);
}
```

Questions

3 Mark Questions

1. What is modularisation? What are its advantages?
2. What is a function? What are the advantages of using functions?
3. Describe the structure of a function in C.
4. What are arguments? What is their purpose?
5. Differentiate between actual and formal parameter.
6. What is meant by a function reference? Summarize the rules that apply to a function call.
7. What is the purpose of return statement? Summarize the rules of governing the use of return statement.
8. What is the purpose of the keyword void? Where this keyword used?
9. What are function prototypes? What is their purpose?
10. Summarize the rules associated with the function prototypes.
11. Explain the different techniques used for passing arguments to a function.
12. Explain how are parameters passed by value. Explain the advantages and disadvantages of this technique.
13. Explain how are parameters passed by reference. Explain the advantages and disadvantages of this technique.
14. Differentiate between the parameter passing mechanisms passing by value and passing by reference. Explain their relative merits and demerits.
15. What is meant by storage class of a variable? Explain the storage-class specifiers in C.
16. What is meant by the scope of a variable within a program? What is the scope of local and global variables?
17. What is an automatic variable? What is its scope?
18. How is an automatic variable defined? Explain its scope and permanence.
19. What is an external variable? What is its scope? Explain its purpose.
20. Differentiate between external variable definition and external variable declaration.
21. How is an external variable defined? Explain its scope and permanence.
22. Differentiate between local and global variables.
23. Differentiate between automatic and external variables.
24. How is a static variable defined? What is its scope and permanence?
25. Differentiate between the internal and external static variables.
26. Differentiate between static and automatic variables.

27. What is the purpose of static variables? Give an example.
28. What is the purpose of register storage class? What benefits are obtained from the use of this storage class?
29. How is register variable defined? What is its scope?
30. Differentiate between automatic and register variables.
31. What is recursion? What are the necessary requirements for solving a problem recursively?

Programming Questions

Draw flowchart and write C program using functions for the following questions.

1. To input a number and to display whether it is even or odd. Use a function that returns 1 if the number is odd, 0 otherwise.
2. To input 3 numbers and to display the biggest among them. Use a function that calculates the larger of two numbers.
3. To input 3 numbers and to display the biggest and smallest among them. Use two functions named max and min. The max function calculates the larger of two numbers and the min function calculates the smaller of two numbers.
4. To check whether a given number is prime or not. Use a function that returns 1 if the numbers is prime, 0 otherwise.
5. To display the prime numbers up to a given number. Use a function that returns 1 if the numbers is prime, 0 otherwise.
6. To display the first ‘n’ prime numbers. Use a function that returns 1 if the numbers is prime, 0 otherwise.
7. To display the first ‘n’ prime numbers. Use a function that returns n^{th} prime number.
8. To display the prime numbers within a range. Use a function that returns 1 if the numbers is prime, 0 otherwise.
9. To check whether a given number is Armstrong number or not. Use a function that returns 1 if the number is Armstrong number, 0 otherwise.
10. To display the first ‘n’ Armstrong numbers. Use a function that returns 1 if the number is Armstrong number, 0 otherwise.
11. To display the Armstrong numbers within a range. Use a function that returns 1 if the number is Armstrong number, 0 otherwise.
12. To check whether a given number is a Fibonacci number or not. Use a function that returns 1 if the number is a Fibonacci, 0 otherwise.
13. To display the Fibonacci numbers 1,1,2,3,5,8, ... up to a given number. . Use a function that returns the n^{th} Fibonacci number.

14. To display the first ‘n’ Fibonacci numbers. Use a function that returns the n^{th} Fibonacci number.
15. To display the Fibonacci numbers within a range. Use a function that returns the n^{th} Fibonacci number.
16. To check whether a given number is a perfect factorial or not. Use a function that returns the factorial of a number.
17. To display the factorial of a number. Use a function that returns the factorial of a number.
18. To sum the series 1!+2!+3!+....+10!. Use a function that returns the factorial of a number.
19. To display the sum of the factorials of the odd numbers from 1 to 10. Use a function that returns the factorial of a number.
20. To display the sum of the factorials of the even numbers from 1 to 10. Use a function that returns the factorial of a number.
21. To sum the series 1!+2!+3!+.... up to the n^{th} term. Use a function that returns the factorial of a number.
22. To sum the Sine series $x-x^3/3!+ x^5/5!-$ .... with an accuracy of 0.0001. Where x is the angle in radian. Use a function that returns the factorial of a number. Also use another function that returns the value of x^n.
23. To sum the Cosine series $1-x^2/2!+ x^4/4!-$ .... with an accuracy of 0.0001. Where x is the angle in radian. Use a function that returns the factorial of a number. Also use another function that returns the value of x^n.
24. To sum the series 1+1/2!+1/3!+.... up to a given term. Use a function that returns the factorial of a number.
25. To input a number and display the number in reverse. Use a recursive function for this purpose.
26. To input a number and to display whether it is a palindrome number. Use a function that returns 1 if the number is palindrome, 0 otherwise.
27. To display 5-digit palindrome numbers. Use a function that returns 1 if the number is palindrome, 0 otherwise.
28. To display palindrome numbers from 1 to 10000. Use a function that returns 1 if the number is palindrome, 0 otherwise.
29. To display the palindrome numbers within a given range. Use a function that returns 1 if the number is palindrome, 0 otherwise.
30. To input 10 numbers and display the smallest and largest among them. Use two functions that return larger and smaller of two numbers.
31. To input ‘n’ numbers and display the smallest and largest among them. Use two functions that return larger and smaller of two numbers.
32. To check whether three sides a, b, and c can form a triangle. If so display its nature (Isosceles, Scalene or Equilateral) and area. The triangle cannot be formed if the any of the side is negative or zero or

if any length is grater than the sum of the other two. Use a function that returns the following values:

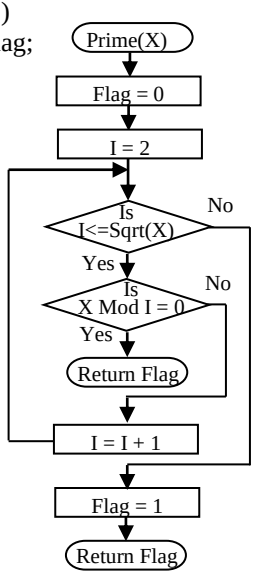
0 – If the values cannot be the sides of a triangle.
1 – If the triangle is equilateral.
2 – If the triangle is isosceles
3 – If the triangle is scalene.

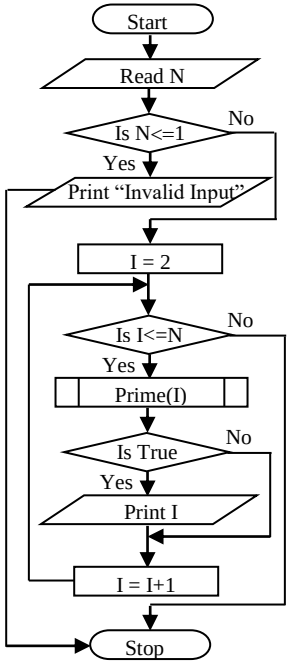
33. To check whether a given year is leap year or not. The program should terminate only when a zero is entered. Use a function that returns 1 if the year is a leap year, 0 otherwise.
34. To display the factorial of numbers from 1 to 20. Use a function that returns the factorial of a number.
35. To display the factorial of odd numbers from 1 to 20. Use a function that returns the factorial of a number.
36. To display the factorial of even numbers from 1 to 20. Use a function that returns the factorial of a number.
37. To check whether a given number is 'Perfect', 'Abundant' or 'Deficient' based on the following. A number is Perfect if the sum of its divisors (excluding the number itself) is equal to the number, e.g.,6. A number is Abundant if the sum of its divisors (excluding the number itself) is grater than the number, e.g., 12. A number is Deficient if it is neither Perfect nor Abundant. Use a function that returns the following values:
    0 – If Perfect
    1 – If Abundant
    2 – If Deficient
38. To classify the numbers within a rage into 'Perfect', 'Abundant' or 'Deficient' based on the following. A number is Perfect if the sum of its divisors (excluding the number itself) is equal to the number, e.g.,6. A number is Abundant if the sum of its divisors (excluding the number itself) is grater than the number, e.g., 12. A number is deficient if it is neither Perfect nor Abundant. Use a function that returns the following values:
    0 – If Perfect
    1 – If Abundant
    2 – If Deficient

Answers to Select Questions

5. /* Program to display Prime numbers up to a given number */

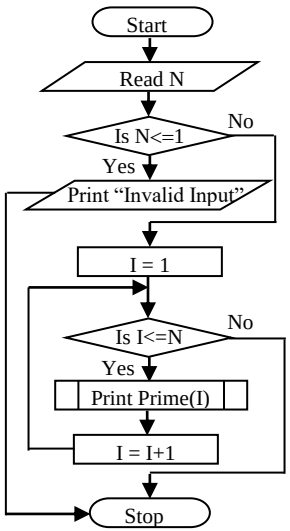
```
#include<stdio.h>
#include<math.h>
int prime(int x);
main()
{  int n,i;
   clrscr();
   printf("Up to which number ? ");
   scanf("%d",&n);
   if(n<0)
      printf("Invalid Input");
   else
   {  printf("The Prime numbers up to %d are:\n",n);
      for(i=2;i<=n;++i)
         if(prime(i))
            printf("%d\t",i);
   }
   getch();
}
int prime(int x)
{  int flag = 0,i;
   for(i=2;i<=sqrt(x);++i)
      if(x%i==0)
         return flag;
   flag = 1;
   return flag;
}
```

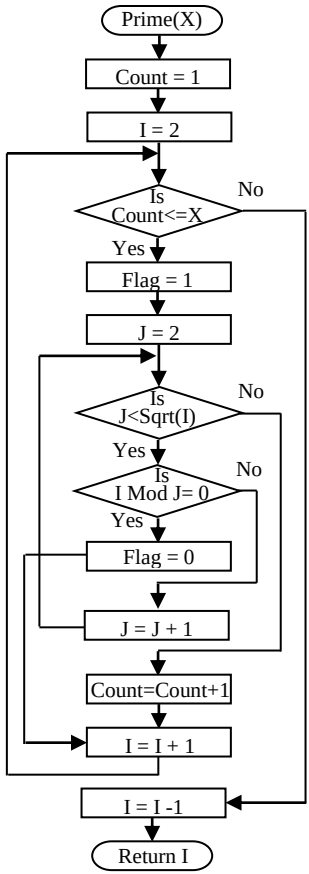




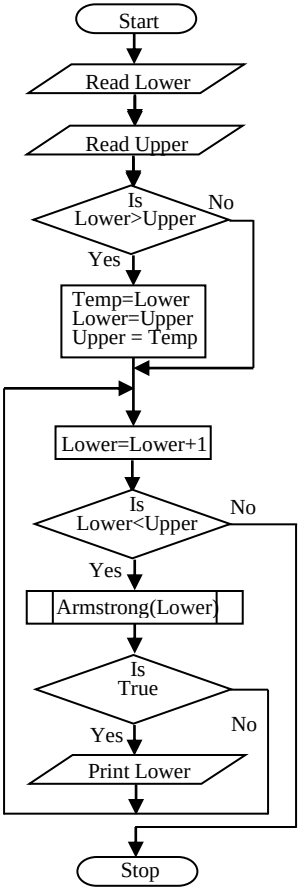
7.

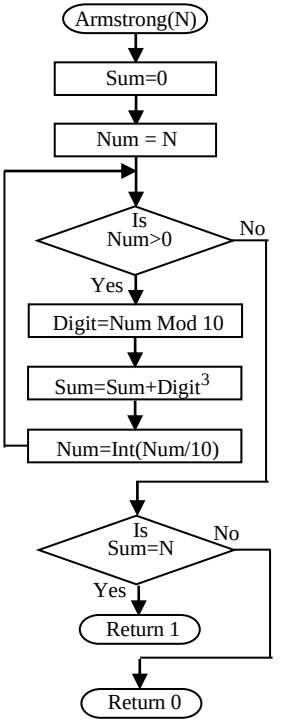
```
/* First 'n' Prime numbers */
#include<stdio.h>
#include<math.h>
int prime(int x);
main()
{  int n,i;
   clrscr();
   printf("Up to which number ? ");
   scanf("%d",&n);
   if(n<0)
      printf("Invalid Input");
   else
   {  printf("The first %d Prime numbers are:\n",n);
      for(i=1;i<=n;++i)
            printf("%d\t",prime(i));
   }
   getch();
}
int prime(int x)
{  int i,j,count=1,flag;
   for(i=2;count<=x;++i)
   {  flag = 1;
      for(j=2;j<=sqrt(i);++j)
         if(i%j==0)
            {  flag = 0;
               break;
            }
      if(flag)
         count++;
    }
   i--;
   return i;
}
```





11.

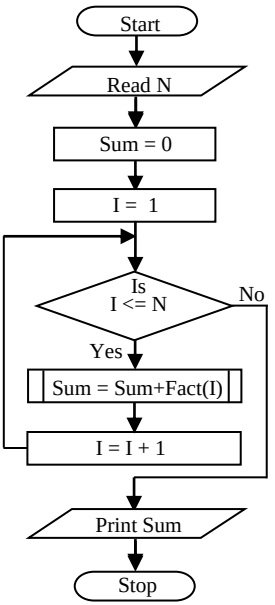


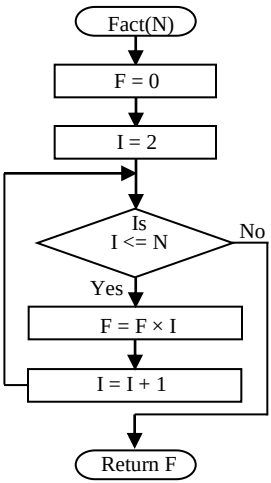



```
/* To display the Armstrong numbers within a range */
#include<stdio.h>
#include<math.h>
int armstrong(int n);
main()
{  int lower,upper,temp;
   clrscr();
   printf("Enter a range \n");
   printf("\tLower Bound ? ");
   scanf("%d",&lower);
   printf("\tUpper Bound ? ");
   scanf("%d",&upper);
   if(lower>upper)
```

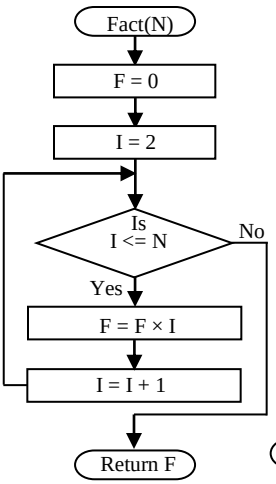
```
  { temp=lower;
    lower=upper;
    upper=temp;
  }
  printf("Armstrong Numbers between %d and %d are:\n",lower,upper);
  for(++lower;lower<upper;++lower)
    if(armstrong(lower))
      printf("%d\t",lower);
  getch();
}
int armstrong(int n)
{ int num,sum=0,digit;
  num=n;
  while(num>0)
  { digit=num % 10;
    sum+=pow(digit,3);
    num/=10;
  }
  if(sum==n)
    return 1;
  else
    return 0;
}
```

21.



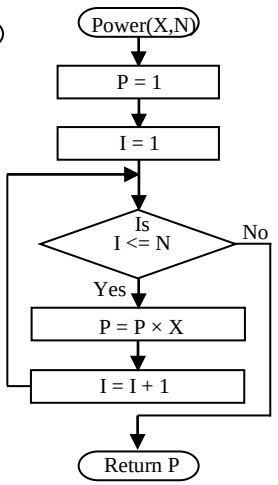


```
/* To display the sum of 1!+2!+3!+....+n! */
#include <stdio.h>
double fact(int n);
main()
{
   int n,i;
   double sum=0.0;
   clrscr();
   printf("Up to which term ? ");
   scanf("%d",&n);
   for(i=1;i<=n;++i)
      sum+=fact(i);
   printf("Sum of the factorials of the numbers up to %d is %g",n,sum);
   getch();
}
double fact(int n)
{  int i;
   double f = 1.0;
      for(i=2;i<=n;++i)
         f*=i;
   return f;
}
```



23.

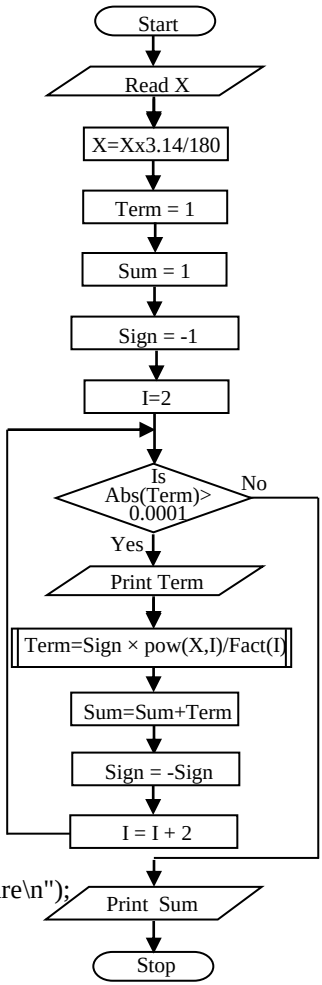
```
/* To Sum the Cosine series */
#include<stdio.h>
#include<math.h>
double fact(int n);
double power(float x, int n);
main()
{
   int i;
   float sign,x,tempx;
   double sum=1.0,term=1.0;
   clrscr();
   printf("Enter the angle in degree : ");
   scanf("%f",&x);
   tempx=x;
   x=x*3.14/180.0;
   sign = -1;
   printf("The terms are:\n");
   for(i=2;fabs(term)>0.0001;i+=2)
```



```
   {  printf("%g\t",term);
      term=sign*power(x,i)/fact(i);

      sum+=term;
      sign = -sign;
   }
   printf("\n\nCos(%g) = %.2f",tempx,sum);
   getch();
}

double fact(int n)
{  int i;
   double f = 1.0;
   for(i=2;i<=n;++i)
      f*=i;
   return f;
}
double power(float x, int n)
{  double p=1,i;
   for(i=1;i<=n;++i)
      p=p*x;
   return p;
}
```

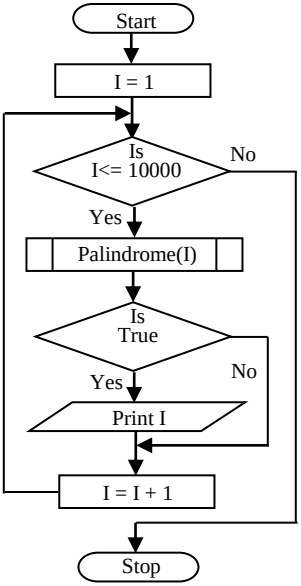


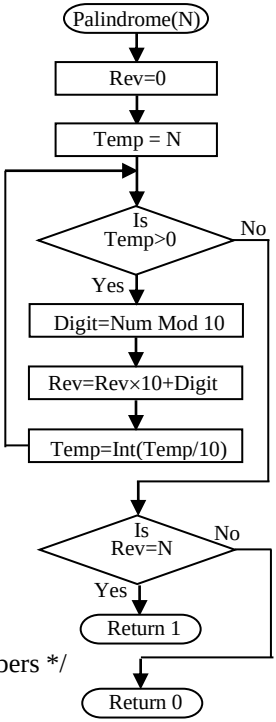
28.

```
/* To display the palindrome numbers */
#include<stdio.h>
int palindrome(int n);
main()
{  int i;
   clrscr();
   printf("Palindrome numbers form 1 to 10000 are\n");
   for(i=1;i<=10000;i++)
      if(palindrome(i))
         printf("%d\t",i);
   getch();
}

int palindrome(int n)
{  int i, rev=0, digit,temp;
   temp = n;
   while(temp>0)
   {  digit = temp % 10;
      rev=rev*10+digit;
```

```
    temp/=10;
  }
  if(rev==n)
    return 1;
  else
    return 0;
}
```



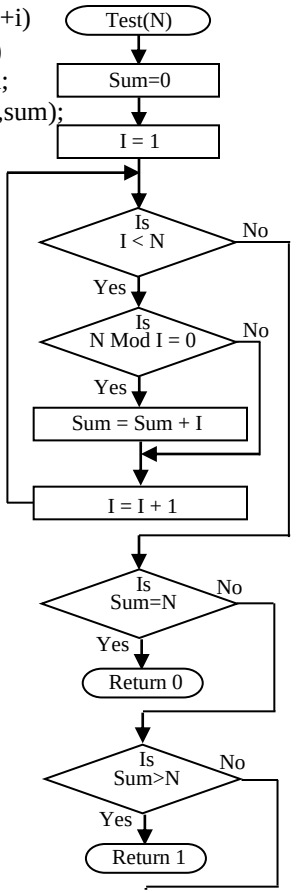


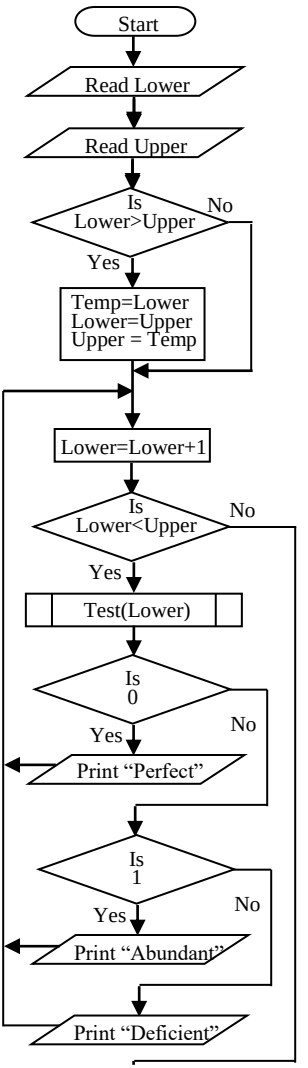
38.

```
/* To display Perfect, Abundant and Deficient numbers */
#include<stdio.h>
int test(int n);
main()
{  int i,lower,upper,temp;
   clrscr();
   printf("Enter a range \n");
   printf("\tLower Bound ? ");
   scanf("%d",&lower);
   printf("\tUpper Bound ? ");
   scanf("%d",&upper);
   if(lower>upper)
   {  temp=lower;
      lower=upper;
      upper=temp;
   }
   printf("Number\tType\n");
   for(++lower;lower<upper;lower++)
      switch(test(lower))
      {  case 0: printf("%d\tPerfect\n",lower);
                 break;
         case 1: printf("%d\tAbundant\n",lower);
```

```
                break;
        case 2: printf("%d\tDeficient\n",lower);
      }
   getch();
}

int test(int n)
{  int i, sum=0;
   for(i=2;i<n;++i)
      if(n%i==0)
         sum += i;
   printf(" %d ",sum);
   getch();
   if(sum==n)
      return 0;
   else
      if(sum>n)
         return 1;
      else
         return 2;
}
```





UNIT VI

STRUCTURED DATA TYPES

6.1. Arrays

Many applications require the processing of multiple data items that have common characteristics (e.g., a set of numerical data, represented by x_1, x_2, x_3, x_4,... , x_n). In such situations, it is often convenient to palace the data items into an array. An array is an identifier that refers to a collection of data items that all have the same name. The data items must all be of the same type (e.g., all integers, all characters, all floats, etc.) and the same storage class. The individual data items are represented by their corresponding array element (i.e., the first data item is represented by the first array element, etc.). The individual array elements are distinguished from one another by the value that is assigned to a subscript (sometimes called index). An individual data item within an array is referred to by specifying the array name followed by one or more subscripts, with each subscript enclosed in square brackets. Each subscript must be expressed as a nonnegative number. The value of the subscript can be expressed as an integer constant, an integer variable or a more complex integer expression. In C, array indices always begin with zero. There are several different ways to recognize arrays (e.g., integer arrays, character arrays, one-dimensional arrays, multi-dimensional arrays)

6.2. Defining a One-Dimensional Array

Arrays are defined in the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression, enclosed in square brackets. The expression is usually written as positive integer constant.

In general terms, a one-dimensional array definition may be expressed as

storage-class data-type array-name[exp1];

where storage-class refers to the storage class of the array, data-type is its data type, array-name is the name of the array and exp1 is a positive-valued integer expressions which indicates the number of array elements. The storage-class is optional; the default values are automatic for arrays that are defined within a function or a block, and external for arrays defined outside of a function.

Several typical one-dimensional array definitions are shown below.

```
int x[100];          /* An 100-element integer array */
char text[80];       /* An 80-element character array */
```

```
static float n[20]; /* A 20-element floating-point static array */
```

It is sometimes convenient to define array size in terms of symbolic constant rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size (e.g., within for loop as well as in array definitions) can be altered simply by changing the value of the symbolic constant.

The array definition can include the assignment of initial values, if desired. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas. The general form is

storage-class data-type array-name[exp1]= {value 1, value 2, ...,value n};

where value 1 refers to the value of the first element, value 2 refers to the value of the second element, and so on.

Several array definitions that include the assignment of initial values are shown below.

```
int num[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float x[5]={0.1, 0,0.2,0.3,0.1};
char colour[8]={‘v’, ‘i’, ‘b’, ‘g’, ‘y’, ‘o’, ‘r’}
```

The results of these initial assignments, in terms of individual array elements are, are as follows.

```
num[0] = 1      x[0] = 0.1    colour[0] = ‘v’
num[1] = 2      x[1] = 0      colour[1] = ‘i’
num[2] = 3      x[2] = 0.2    colour[2] = ‘b’
num[3] = 4      x[3] = 0.3    colour[3] = ‘g’
num[4] = 5      x[4] = 0.1    colour[4] = ‘y’
num[5] = 6                    colour[5] = ‘o’
num[6] = 7                    colour[6] = ‘r’
num[7] = 8
num[8] = 9
num[9] = 10
```

The array size need not be specified explicitly when initial values are indicated as a part of array definition. With a numerical array, the array size will automatically be set equal to the number initial values included within the definition.

For example, consider the following array definitions.

```
int num[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float x[]={0.1, 0,0.2,0.3,0.1}
```

Thus, num will be a 10-element integer array and x will be a 5-element floating-point array. These declarations will result in the same assignment of initial values to array elements, as shown in the previous example.

Strings can be assigned to character type arrays. To do so, the array is usually written without explicit size specification. The proper array size

will be assigned automatically. When a string is assigned to a character array, a null character is automatically added at the end of every string.
For example, consider the following character array definition.

```
message[]= “I like C”
```

The elements of this 9-element character array are

```
message[0]=‘I’   message[3]=‘i’   message[6]=‘ ’
message[1]=‘ ’   message[4]=‘k’   message[7]=‘C’
message[0]=‘l    message[5]=‘e’   message[9]=‘\0’
```

6.3. Multidimensional Arrays

Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on. In general terms, a multidimensional array definition can be written as

```
storage-class data-type array-name[exp1] [exp2]... [expn];
```

where storage-class refers to the storage class of the array, data-type is the data type, array-name is the name of the array, and exp1, exp2, ..., expn are positive valued integer expressions that indicate the number of array elements associated with the each subscript. The storage-class is optional; the default values are automatic for arrays that are defined within a function or a block, and external for arrays defined outside of a function.
Examples of multidimensional array definitions are shown below.

```
float table[100][5];        /*Floating-point array having 100 rows and
                              5 columns, hence 500 elements */
char names[50][20];         /*String array having 50 strings of 20
                              characters each */
int tables[50][100][5];     /* 50 tables each having 100 rows and 5
                              columns */
```

Multidimensional array definitions can include the assignment of initial values, if desired. Initialisation of a two-dimensional array is done by specifying the elements in row major order (i.e., with the elements of the first row in a sequence, followed by those of the second, and so on. An example is given below.

```
int matrix[3][4]= {
                        { 1, 2, 3, 4 },
                        { 5, 6, 7, 8 },
```

```
        { 9, 0, 1, 2 }
    };
```

The first subscript can be omitted from the array definition when initial values are indicated as a part of array definition. For example, the declaration given below, wherein the number of rows is not explicitly specified, is equivalent to the previous one.

```
int matrix[][4] = {
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 0, 1, 2 }
    };
```

Also, the inner brackets can be omitted, giving the numbers in one continuous sequence. For example,

```
int matrix[3][4]={1,2,3,4,5,6,7,8,9,0,1,2};
```

has the same effect as the previous example, but is not readable as before.

6.4. Processing an Array

Single operations, which involve entire arrays, are not permitted in C. Thus, the array processing must be carried out on element-by-element basis. This is usually accomplished within a loop. The number of passes through the loop will therefore equal the number of array elements to be processed.

For example, the following program will read a set of numbers and calculate their average. It then calculates and displays the deviation of each number about the average.

```
/* To display deviation of each number about their average */
#include<stdio.h>
main()
{  int i,n;
   float num[25],sum=0,avg;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   for(i=0;i<n;++i)
   {  printf("Enter Number - %d : ",i+1);
      scanf("%f",&num[i]);
      sum+=num[i];
   }
   avg=sum/n;
   printf("Number\t\tAverage\t\tDeviation\n");
```

```
        printf("------\t\t-------\t\t---------\n");
        for(i=0;i<n;++i)
           printf("%g\t\t%g\t\t%+g\n",num[i],avg,num[i]-avg);
        getch();
     }
```

The following program that adds two matrices illustrates the processing of a two-dimensional array.

```
/* Matrix addition */
#include<stdio.h>
main()
{
  int a[10][10],b[10][10],c[10][10];
  int i,j,row,col;
  clrscr();
  printf("How many rows? ");
  scanf("%d",&row);
  printf("How many columns? ");
  scanf("%d",&col);
  printf("Enter Your First Matrix\n");
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
    {  printf("Element(%d,%d) ? ",i+1,j+1);
       scanf("%d",&a[i][j]);
    }
  clrscr();
  printf("Enter Your Second Matrix\n");
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
    {  printf("Element(%d,%d) ? ",i+1,j+1);
       scanf("%d",&b[i][j]);
    }
  clrscr();
  for(i=0;i<row;++i)
    for(j=0;j<col;++j)
       c[i][j]=a[i][j]+b[i][j];
  printf("\t\tFirst Matrix\n");
  printf("\t\t------------\n\t\t");
  for(i=0;i<row;++i)
  {  for(j=0;j<col;++j)
       printf("%d   ",a[i][j]);
    printf("\n\t\t");
  }
  printf("Second Matrix\n");
```

```
   printf("\t\t------------\n\t\t");
   for(i=0;i<row;++i)
   {  for(j=0;j<col;++j)
         printf("%d   ",b[i][j]);
      printf("\n\t\t");
   }
   printf("Sum Matrix\n");
   printf("\t\t------------\n\t\t");
   for(i=0;i<row;++i)
   {  for(j=0;j<col;++j)
         printf("%d   ",c[i][j]);
      printf("\n\t\t");
   }
   getch();
}
```

6.5. Passing Arrays to Function

An entire array can be passed to a function as an argument. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written as an array within the formal argument declaration. The formal argument declaration of arrays within the function definition must include explicit size specification in all subscript positions, except the first. The size specification must be consistent with the corresponding size specifications in the calling program. The first subscript position may be written as an empty pair of square brackets. The corresponding function prototypes must be written in the same manner. Thus, when declaring a one-dimensional array as formal argument, the array name is written with a pair of empty square brackets and the size of the array is not specified within the formal argument declaration.

When an array is passed to a function, the array name is interpreted as the address of the first array element. This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first array element. Therefore, the arrays arguments are passed to a function by reference rather than by value. Therefore, any array element can be accessed from within the function. If an array element is altered within the function, the alteration will be recognized in the calling portion of the program.

The following program that sorts a set of numbers illustrates the passing of a one-dimensional array to a function.

```
/* To sort a set of numbers using functions*/
#include<stdio.h>
void readnum(float a[], int n);
void printnum(float a[], int n);
void sort(float a[], int n);
main()
{  int n,num[50];
   clrscr();
   printf("How many Numbers ? ");
   scanf("%d",&n);
   readnum(num,n);
   clrscr();
   printf("\nThe numbers you entered are:\n");
   printnum(num,n);
   sort(num,n);
   printf("\nThe sorted numbers are:\n");
   printnum(num,n);
   getch();
   }

/* Function to read a set of numbers */
void readnum(float a[], int n)
{  int i;
   for(i=0;i<n;++i)
   {  printf("Enter Number - %d : ",i+1);
      scanf("%f",&a[i]);
   }
   return;
}

/* Function to display a set of numbers */
void printnum(float a[], int n)
{  int i;
   for(i=0;i<n;++i)
      printf("%g\t",a[i]);
   return;
}

/* Function to sort a set of numbers using bubble sort */
void sort(float a[], int n)
{  int i,j;
   float temp;
   for(i=0;i<n-1;++i)
      for(j=0;j<(n-1)-i;++j)
```

```
            if(a[j]>a[j+1])
            {  temp = a[j];
               a[j]=a[j+1];
               a[j+1]=temp;
            }
      return;
}
```

The sort function can be rewritten using selection sort as shown below.

```
/* Function to sort a set of numbers using Selection Sort */
void sort(float a[], int n)
{  int i,j;
   float temp;
   for(i=0;i<n-1;++i)
      for(j=i+1;j<n;++j)
         if(a[i]>a[j])
         {  temp = a[i];
            a[i]=a[j];
            a[j]=temp;
         }
   return;
}
```

The following program that determines the product of two matrices illustrates the passing of two-dimensional arrays to a function.

```
/* Matrix multiplication */
#include<stdio.h>
void readmat(int x[][10], int r, int c);
void printmat(int x[][10], int r, int c);
void multiplymat(int x[][10],int y[][10], int z[][10],int r, int c, int d);
main()
{  int mat1[10][10],mat2[10][10],mat3[10][10];
   int i,j,row1,col1,row2,col2;
   clrscr();
   printf("\tEnter the Order of the first matrix\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row1);
   printf("\t\tNumber of columns ? ");
   scanf("%d",&col1);
   printf("\n\n\tEnter the Order of the second matrix\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row2);
   printf("\t\tNumber of columns ? ");
   scanf("%d",&col2);
   if(col1 != row2)
```

```
      printf("Sorry ! Multiplication is not possible");
   else
   {  clrscr();
      printf("Enter the elements of the first matrix :\n");
      readmat(mat1,row1,col1);
      clrscr();
      printf("Enter the elements of the second matrix :\n");
      readmat(mat2,row2,col2);
      clrscr();
      printf("\tFirst Matrix\n");
      printf("\t------------\n");
      printmat(mat1,row1,col1);
      printf("\n\tSecond Matrix\n");
      printf("\t-------------\n");
      printmat(mat2,row2,col2);
      multiplymat(mat1,mat2,mat3,row1,col2,col1);
      printf("\n\tProduct Matrix\n");
      printf("\t--------------\n");
      printmat(mat3,row1,col2);
   }
  getch();
}

/* Function for reading matrix */
void readmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
      {  printf("Element (%d,%d) ? ",i+1,j+1);
         scanf("%d",&x[i][j]);
      }
   return;
}

/* Function for displaying matrix */
void printmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<c;++j)
         printf("%6d",x[i][j]);
      printf("\n");
   }
   return;
}
```

```
/* Function for matrix multiplication */
void multiplymat(int x[][10],int y[][10],int z[][10],int r,int c, int d)
{  int i,j,k;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
      {  z[i][j]=0;
         for(k=0;k<d;++k)
            z[i][j]=z[i][j]+x[i][k]*y[k][j];
      }
   return;
}
```

6.6. Strings

In most applications, a string of characters rather than a single character is necessary. As an individual character variable can store only one character, we need an array of characters to store strings. Thus, in C, a string is stored in an array of characters. Each character in the string occupies one location in an array. The null character '\0' is put after the last character. This is done so that program can tell when the end of a string has been reached. For example, the string "Enter a Number:" is stored as follows

'E'	'n'	't'	'e'	'r'	' '	'a'	' '	'N'	'u'	'm'	'b'	'e'	'r'	':'	'\0'

Since the string has 15 characters (including the space), it requires an array of at least, size 16 to store it.

Thus, in C, a string is a one-dimensional array of characters terminated by a null character. The terminating null character is important. In fact, a string not terminated by a '\0' is not really a string, but merely a collection of characters.

Some applications require that the characters within a string be processed individually. While some other applications require that the string be processed as complete entities. These types of problems can be simplified considerably through the use of special string oriented library functions. For example, most C compilers include library functions that allow string to be compared, copied or concatenated (i.e., combined, one behind another). Other library functions permit operations on individual characters within strings; e.g., they allow individual characters to be located within strings, and so on.

The most commonly used string functions are given below.

strlen(str)	Returns the length of the null-terminated string pointed to by str. The null terminator is not counted.
strlwr(str)	Converts str to all lowercase
strupr(str)	Converts str to all uppercase
strcat(str1,str2)	Concatenates a copy of the string pointed to by str2 to the string pointed to by str1 and terminates str1 with a null. The null terminator originally ending str1 is overwritten by the first character of str2. The str2 is untouched by the operation.
strncat(str1,str2,n)	Concatenates not more than n characters of the string pointed to by str2 to the string pointed to by str1 and terminates str1 with a null. The null terminator originally ending str1 is overwritten by the first character of str2. The str2 is untouched by the operation.
strcpy(str1,str2)	Copies the string pointed to by str2 to the string pointed to by str1 and terminates str1 with a null. The str2 must be a pointer to a null terminated string.
strncpy(str1,str2,n)	Copies up to n characters from the string pointed to by str2 to the string pointed to by str1 and terminates str1 with a null. The str2 must be a pointer to a null terminated string.
strcmp(str1,str2)	Compares str1 and str2 lexicographically. Returns a negative value if str1<str2; 0 if str1 and str2 are identical; and positive value if str1>str2.
strcmpi(str1,str2)	Compares str1 and str2 lexicographically, without regard to case. Returns a negative value if str1<str2; 0 if str1 and str2 are identical; and positive value if str1>str2.
strncmp(str1,str2)	Compares str1 and str2 lexicographically. Returns a negative value if str1<str2; 0 if str1 and str2 are identical; and positive value if str1>str2.
stricmp(str1,str2)	Equivalent to strcmpi.
strncmp(str1,str2,n)	Compares at most n characters of str2 to str1, lexicographically. Returns a value that is < 0 if str1 is less than str2; == 0 if s1 is the same as str2; > 0 if s1 is greater than s2.
strncmpi(str1,str2,n)	Compares at most n characters of str2 to str1, lexicographically, by ignoring case. Returns a value

	that is < 0 if str1 is less than str2; == 0 if s1 is the same as str2; > 0 if s1 is greater than s2.
strnicmp(str1,str2,n)	Equivalent to strncmpi.
strchr(str,c)	Finds c in str. Returns a pointer to the first occurrence of the character c in str; if c does not occur in str, strchr returns NULL.
strchr(str,c)	Finds the last occurrence of c in str. Returns a pointer to the last occurrence of the character c, or NULL if c does not occur in str.
strstr(str1,str2)	Finds the first occurrence of substring str2 in str1. Returns a pointer to the element in str1 that contains s2 (points to s2 in s1), or NULL if s2 does not occur in s1.
strset(str,c)	Sets all characters in str to c. Quits when the first null character is found. Returns a pointer to str.
strnset(str,c,n)	Sets the first n characters of str to c. Stops when n characters are set or a NULL is found. Returns pointer to s.
strrev(str)	Reverses all characters in str (except for the terminating null). Returns a pointer to the reversed string.

For example, the following program will read and sort a list of names. The strings are stored in a two-dimensional character array. Each string will be stored in a separate row within the array. While reading a one-dimensional array using the scanf function, the array name should not precede an ampersand.

```
/* Sort a set of name using Bubble Sort */
#include<stdio.h>
#include<string.h>
main()
{ int i,j,n;
   char name[50][20],temp[20];
   clrscr();
   printf("How many names ? ");
   scanf("%d",&n);
   for(i=0;i<n;++i)
   {  printf("Enter Name - %d : ",i+1);
      scanf("   %[^\n]",name[i]);
      /* Blank space in the beginning of the format string is required */
   }
   for(i=0;i<n-1;++i)
      for(j=0;j<(n-1)-i;++j)
         if(strcmp(name[j],name[j+1]) > 0)
```

```
        {  strcpy(temp,name[j]);
           strcpy(name[j],name[j+1]);
           strcpy(name[j+1],temp);
        }
  printf("The sorted Names are:\n");
  for(i=0;i<n;++i)
     printf("%s\n",name[i]);
  getch();
  }
```

6.7. Pointers

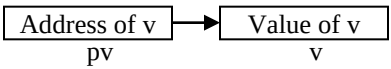
Pointer is a variable that represents the location of a data item, such as a variable or an array element. Within computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent bytes). The number of memory cells required to store a data item depends on the type of the data item. For example, a single character will typically be stored in one byte of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes, and a double-precision quantity may require eight contiguous bytes.

Suppose v is variable, that represents some particular data item. The compiler will automatically assign memory cells to this data item. The data item can then be accessed if we know the address of the first memory cell. The address of v's memory location can be determined by the expression &v, where & is a unary operator, called the address operator, that evaluates the address of its operand.

Now let us assign the address of v to another variable, pv. Thus,

pv = &v;

This new variable is called a pointer to v, since it points to the location where v is stored in the memory. Thus, pv is referred to as a pointer variable. The relationship between pv and v is illustrated below



The data item represented by v (i.e., the data item stored in v's memory cells) can be accessed by the expression *pv, where * is a unary operator, called the indirection operator, that operates only on a pointer variable. Therefore, *pv and v both represent the same data item.

The address operator (&) and the indirection operators (*) are unary operators and they are the members of the same precedence group as the other unary operators. The address operator (&) must act up on operands that associated with unique addresses, such as an ordinary variable or single array element. Thus, the address operators cannot act upon arithmetic expressions.

The indirection (*) operator can only act upon operands that are pointers (e.g., pointer variables)

6.8. Pointer Declaration

Pointer variables, like all other variables, must be declared before they may be used in a C program. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies that the variable is a pointer. The data type that appears in the declaration refers to the object of the pointer, i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as

data-type *ptvar;

where ptvar is the name of the pointer variable, and data-type refers to the data-type of the pointer's object.

For example, a C program contains the following declarations.

```
int u,*pu;
float v, *pv;
```

The first line declares u to be an integer type variable and pu to be a pointer variable whose object is an integer quantity. The second line declares v to be a floating-point type variable and pv to be a pointer variable whose object is a floating-point quantity.

Within a variable declaration, a pointer variable can be initialised by assigning it the address of another variable. However, the variable whose address is assigned to the pointer variable must have been declared earlier in the program.

For example, a C program contains the following declarations.

```
float v;
float *pv=&v;
```

The first line declares v to be floating-point type variable and the second line declares pv to be a pointer variable whose object is a floating-point quantity. In addition, the address of v is initially assigned to pv.

The following program illustrates the use of pointer variables.

```
#include<stdio.h>
main()
{  int v = 5;
      int *pv=&v;
      printf("\n*pv=%d,v=%d,&v=%X and pv=%X",*pv,v,&v,pv);
}
```

When this program is executed, the following output is generated.

```
*pv=5,v=5,&v=FFD2 and pv=FFD2
```

The first line declares v to be an integer variable that represents the value 5. The second line declares pv to be a pointer variable whose object is an integer quantity. In addition, the address of v is initially assigned to pv. The address of v is determined automatically by the compiler as FFD2 (hexadecimal). The pointer pv is assigned this value; hence, pv also represents the address FFD2.

6.9 Passing Pointers to a Function

Pointers can be passed to function as arguments and can also be returned to the calling function. This allows the data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function. . This form of argument passing is referred to passing by reference.

When pointers are used as actual arguments, the corresponding formal arguments must be declared as pointer variable in the formal argument declaration within the function. The usage of pointer arguments is illustrated in the following examples.

```
#include <stdio.h>
void max(int *px, int *py);
main()
{  int x,y;
   clrscr();
   printf("First No  : ");
   scanf("%d",&x);
   printf("Second No : ");
   scanf("%d",&y);
   max(&x,&y);
   getch();
}

void max(int *px, int *pv)
{  int k;
   k=(*px>*py)?*px:*py;
   printf("Largest among %d and %d is %d ",*px,*py,k);
   return;
}
```

6.10. Operations on Pointers

The C language allows arithmetic operations to be performed on pointer variables. It is, however, the responsibility of the programmer to see that the result obtained by performing pointer arithmetic is the address of relevant and meaningful data. The arithmetic operators available for use with pointers can be classified as:

Unary operators: ++ (increment) and –– (decrement)

Binary operators: + (addition) and – (subtraction)

The C compiler takes the size of the data type being pointed to into account, while performing arithmetic operations on a pointer. For example, if a pointer to an integer is incremented using the ++ operator, then the address contained in the pointer is incremented by two and not one, assuming that an integer occupies two bytes in memory. Similarly, incrementing a pointer to a float causes the initial address contained in the float pointer to be actually incremented by 4 and not 1 (if the size of the float variable is 4 bytes on machine). In general, a pointer to some data type, d_type (where d_type may be int, char, float, double or any user defined data type), when incremented by an integral value i will result in the following value:

(current address in pointer)+i×sizeof(d_type)

to be the new incremented pointer value. This concept applies to all arithmetic operations performed on pointer variables. Pointer arithmetic becomes very significant for accessing and processing array element efficiently. Pointer arithmetic cannot be performed on void pointers, since they have no type associated with them.

The following program illustrates the effect of incrementing pointer variables.

```
#include<stdio.h>
main()
{  char c, *pc = &c;
   int i, *pi = &i;
   float f, *pf = &f;
   double d=0.0, *pd = &d;
   clrscr();
   printf("pc = %X, pc+1 = %X, pc+2 = %X\n", pc,pc+1,pc+2);
   printf("pi = %X, pi+1 = %X, pi+2 = %X\n", pi,pi+1,pi+2);
   printf(“pf = %X, pf+1 = %X, pf+2 = %X\n", pf,pf+1,pf+2);
   printf("pd = %X, pd+1 = %X, pd+2 = %X\n", pd,pd+1,pd+2);
```

```
   getch();
}
```

The execution of this program results in the following output.

```
pc = FFC1, pc+1 = FFC2, pc+2 = FFC3
pi = FFC2, pi+1 = FFC4, pi+2 = FFC6
pf = FFC4, pf+1 = FFC8, pf+2 = FFCC
pd = FFCA, pd+1 = FFD2, pd+2 = FFDA
```

Note that any integer constant that is added to or subtracted form the pointer will be interpreted differently. Each integer value will represent an equivalent number of individual bytes in the case of character pointer, a corresponding number of two-byte multiples in the case of integer pointer, a corresponding number of four-byte multiples in the case of floating-point pointer, or a corresponding number of eight-byte multiples in the case of double precision pointer.

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 bytes separating the corresponding array elements. This is illustrated in the following example.

```
#include<stdio.h>
main()
{  int n[10]={100,101,102,103,104,105,106,107,108,109};
   int *px=&n[0], *py=&n[5];
   clrscr();
   printf("px = %X and py = %X\npy-px = %X",px,py,py-px);
   getch();
}
```

Execution of the program results in the following output.

```
px = FFC0 and py = FFCA
py-px = 5
```

The first line of the output indicates that address of n[0] is FFC0 and the address of n[5] is FFCA. The difference between these two hexadecimal numbers is 10 (decimal). Thus, a[5] is stored at an address which is 10 bytes beyond the address of n[0]. Since each integer quantity occupies two bytes, we would expect the difference between py and px to be 10/2 = 5.

Pointer variables can be compared provided both variables are of the same data type. The comparisons can test for equality or inequality. Some examples are:

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

The permissible operations on pointer are summarized below:

1. A pointer variable can be assigned the address of an ordinary variable (e.g., pv=&v).

2. A pointer variable can be assigned the value of another pointer variable (e.g., pv=px).
3. A pointer variable can be assigned a null (zero) value (e.g., pv=0).
4. An integer quantity can be added to or subtracted from a pointer variable (e.g., pv+3, ++pv).
5. One pointer variable can be subtracted from another provided both pointers point to the element of the same array.
6. Two pointer variables can be compared provided both pointers point to objects of the same data type.

6.11. Pointers and One-Dimensional Arrays

The elements of an array can be efficiently accessed by using pointers. An array name is really a pointer to the first element in the array. Thus, the name of the array can be used effectively as any other pointer to the array in accessing the elements of the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. The address of the second array element can be written as &x[1] or as (x+1), and so on. In general, the address of array element (i+1) can be expressed as either &x[i] or (x+i). Thus, there are two different ways to write the address of any array element: one can write the actual array element, preceded by an ampersand; or by writing a pointer expression in which the subscript is added to the array name.

In the pointer expression (x+i), x represents an address and i represent an integer quantity. The compiler takes the size of the data type being pointed by the array name into account, while adding the integer quantity i to the pointer x. While writing the address of an array element in the form of (x+i), however, the user need not be concerned with the number of memory cells associated with each type of array element; the C compiler adjusts for this automatically. One needs to specify the address of the first array element (i.e., the name of the array) and the number of array elements beyond the first (i.e., 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 i^{th} element of x, x[i] and *(x+i) both represent the contents of that address, i.e., the value of the i^{th} element of x.

The following program illustrates the two different ways to access one-dimensional array elements.

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

```
{  int i, x[10]={100,101,102,103,104,105,106,107,108,109};
   clrscr();
   for(i=0;i<10;++i)
   printf("x[%d] = %d\t*(x+%d) = %d\t&x[%d] = %x\t(x+%d) = %x\n",
                                  i,x[i],i,*(x+i),i,&x[i],i,(x+i));
   getch();
}
```

Execution of this program results in the following output.

```
x[0] = 100   *(x+0) = 100  &x[0] = FFC0   (x+0) = FFC0
x[1] = 101   *(x+1) = 101  &x[1] = FFC2   (x+1) = FFC2
x[2] = 102   *(x+2) = 102  &x[2] = FFC4   (x+2) = FFC4
x[3] = 103   *(x+3) = 103  &x[3] = FFC6   (x+3) = FFC6
x[4] = 104   *(x+4) = 104  &x[4] = FFC8   (x+4) = FFC8
x[5] = 105   *(x+5) = 105  &x[5] = FFCA   (x+5) = FFCA
x[6] = 106   *(x+6) = 106  &x[6] = FFCC   (x+6) = FFCC
x[7] = 107   *(x+7) = 107  &x[7] = FFCE   (x+7) = FFCE
x[8] = 108   *(x+8) = 108  &x[8] = FFD0   (x+8) = FFD0
x[9] = 109   *(x+9) = 109  &x[9] = FFD2   (x+9) = FFD2
```

The output clearly illustrates the distinction between x[i], which represents the value of the i^{th} array element, and &x[i], which represents its address. From the output, it can also be seen that the value of the i^{th} array element can be represented by either x[i] or *(x+i), and the address of the i^{th} element can be represented by either &x[i] or (x+i).

From the above discussion, we have seen that an array can be accessed and processed in two ways: by using array name followed by subscripts in square brackets and by using pointer expressions. These two forms are interchangeable. Hence, either form can be used in any particular application. The choice depends on individual preferences. Whereas the array version is more readable, the pointer version is more efficient. In the array version, it is clear that at each step we are looking at the n^{th} element of the array. This is not so obvious in the pointer version. However, evaluation of array version requires the evaluation of subscript n, which is then converted into address of the element n. The pointer version deals with address directly.

The following program illustrates the use of pointers to access and process individual elements of a one-dimensional array.

```
/* To sort a set of numbers using functions*/
#include<stdio.h>
#include<string.h>
void readnum(int *a, int n);
void printnum(int *a, int n);
void sort(int *a, int n);
main()
```

```
{  int n, num[50];
   clrscr();
   printf("How many Numbers ? ");
   scanf("%d",&n);
   readnum(num,n);
   clrscr();
   printf("\nThe numbers you entered are:\n");
   printnum(num,n);
   sort(num,n);
   printf("\nThe sorted numbers are:\n");
   printnum(num,n);
   getch();
   }

/* Function to read a set of numbers */
void readnum(int *a, int n)
{  int i;
   for(i=0;i<n;++i)
   {  printf("Enter Number - %d : ",i+1);
      scanf("%d",a+i);
   }
   return;
}

/* Function to display a set of numbers */
void printnum(int *a, int n)
{  int i;
   for(i=0;i<n;++i)
      printf("%d\t",*(a+i));
   return;
}

/* Function to sort a set of numbers using selection sort */
void sort(int *a, int n)
{  int i,j,temp;
   for(i=0;i<n-1;++i)
      for(j=i+1;j<n;++j)
         if(*(a+i)>*(a+j))
         {  temp = *(a+i);
            *(a+i)=*(a+j);
            *(a+j)=temp;
         }
   return;
}
```

6.12. Pointers and Multidimensional Arrays

A two-dimensional array 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. Thus, a two-dimensional array declaration can be written as

data-type (*ptvar)[exp2];

rather than

data-type array-name [exp1][exp2];

This concept can be generalized to higher-dimensional arrays; that is,

data-type (*ptvar)[exp2][exp3].....[expn];

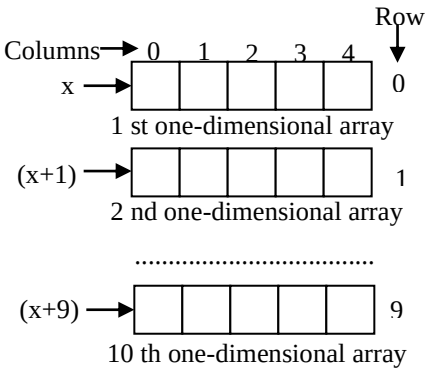
replaces

data-type array-name [exp1][tvar)[exp2][exp3].....[expn];

In these declarations data-type refers to the data type of the array, ptrvar is the name of the pointer variable, array-name is the corresponding array name, and exp1, exp2, ... expn are positive valued integer expressions that indicates the maximum number of array elements associated with each subscript. The parentheses that surround the array name and the preceding asterisk must be present. Without the parentheses, we would be defining an array of pointers rather than a pointer to a group of arrays, since the square brackets and parentheses are normally evaluated from right to left.

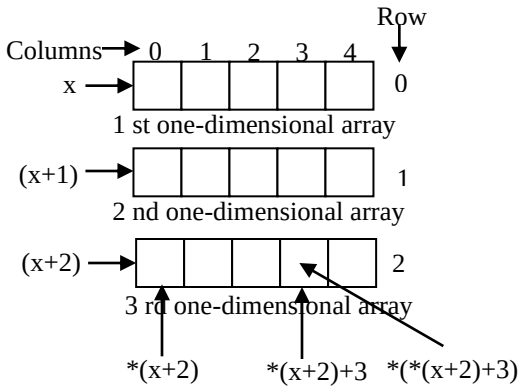
For example, suppose x is a two dimensional array having 4 rows and 3 columns. We can declare x as int (*x)[5] rather than int x[10][5].

In the first declaration, x is defined to be a pointer to a group of contiguous, one-dimensional, 5 element integer arrays. Thus, x points to the first 5-element array, which is actually the first row (row 0) of the original two-dimensional array. Similarly, (x+1) points to the second 5-element array, which is the second row (row 1) of the original two-dimensional array, and so on, as illustrated below.



An individual array element within a multi-dimensional array can be accessed by the repeated use of indirection operator. The following example illustrates the use of indirection operator.

Suppose x is a two-dimensional array having 10 rows and 5 columns, as declared in the previous example. The item in row 2 and column 3 can be accessed by writing either x[2][3] or *(*(x+2)+3). Note that (x+2) is a pointer to row 2. Therefore, the object of this pointer *(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 3 to this pointer. Hence (*(x+2)+3) is a pointer to element 3 (i.e., the fourth element) in row 2. The object of this pointer, *(*(x+2)+3), refers to the item in column 3 of row 2, which x[2][3]. This relationship is illustrated below.



For example, the following program that determines the sum of two matrices illustrates the use pointers to access and process individual array elements of a two-dimensional array.

```
/* To add two matrices */
#include<stdio.h>
void readmat(int *x[10], int r, int c);
void printmat(int *x[10], int r, int c);
void addmat(int *x[10],int *y[10], int *z[10],int r, int c);
main()
{  int mat1[10][10],mat2[10][10],mat3[10][10],row,col;
   clrscr();
   printf("\tEnter the Order of the matrices\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row);
   printf("\t\tNumber of columns ? ");
```

```
   scanf("%d",&col);
   clrscr();
   printf("Enter the elements of the first matrix :\n");
   readmat(mat1,row,col);
   clrscr();
   printf("Enter the elements of the second matrix :\n");
   readmat(mat2,row,col);
   clrscr();
   printf("\tFirst Matrix\n");
   printf("\t------------\n");
   printmat(mat1,row,col);
   printf("\n\tSecond Matrix\n");
   printf("\t-------------\n");
   printmat(mat2,row,col);
   addmat(mat1,mat2,mat3,row,col);
   printf("\n\tSum Matrix\n");
   printf("\t--------------\n");
   printmat(mat3,row,col);
  getch();
}

/* Function for reading matrix */
void readmat(int *x[10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
      {  printf("Element (%d,%d) ? ",i+1,j+1);
         scanf("%d",*(x+i)+j);
      }
   return;
}

/* Function for displaying matrix */
void printmat(int *x[10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<c;++j)
         printf("%6d",*(*(x+i)+j));
      printf("\n");
   }
   return;
}

/* Function for matrix addition */
```

```
void addmat(int *x[10],int *y[10],int *z[10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
           *(*(z+i)+j)=*(*(x+i)+j)+*(*(y+i)+j);
   return;
}
```

6.13. Dynamic Memory Allocation

Since an array name is actually a pointer to the first element within the array, it is possible to define the array as a pointer variable rather than as a conventional array. Syntactically, the two definitions are equivalent. However, in the conventional array definition 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 pointer variables to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as dynamic memory allocation.

In C, storage can be allocated dynamically by using the standard library functions malloc and calloc. In order to use these functions, your program must be preceded by the header line #include<stdlib.h>. The malloc function is called by writing malloc(size). The malloc function allocates 'size' bytes of memory and returns a pointer to the first byte. The storage is not initialised. If it is unable to find the requested amount of memory, malloc returns NULL.

The library function calloc is invoked by writing calloc(num,size). The calloc function allocates num×size bytes of memory and returns a pointer to first byte. All bytes returned are initialised to 0. If it is unable find the requested amount of memory, calloc returns NULL.

The usage of malloc and calloc functions are illustrated below:

Suppose x is a one-dimensional 10-element array of integers. It is possible to define x as a pointer variable rather than array. Thus, we can write int *x rather than int x[10]. 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, the library functions malloc or calloc can be used, as shown below.

```
x = (int *)malloc(10*sizeof(int))
x = (int *)calloc(10,sizeof(int))
```

The argument of malloc is 10*sizeof(int). The operator sizeof returns the number of bytes needed to store a data of type specified. The function malloc

now returns the address where these bytes can be stored. the cast operator (int *) casts this address to be of type pointer to an integer. The calloc function will also perform the same task as malloc. Unlike the malloc, the calloc accepts two arguments. One important difference between malloc and calloc is that calloc initialises all bytes in the allocated block to zero (hence the name calloc - clear and allocate).
The following program that sorts a list of numbers illustrates the dynamic memory allocation. Note that one-dimensional array is defined as a pointer and memory is dynamically allocated by using calloc function.

```
/* To sort a set of numbers using functions*/
#include<stdio.h>
#include<string.h>
void readnum(int *a, int n);
void printnum(int *a, int n);
void sort(int *a, int n);
main()
{  int n,*num;
   clrscr();
   printf("How many Numbers ? ");
   scanf("%d",&n);
   num = (int *) calloc(n,sizeof(int));
   readnum(num,n);
   clrscr();
   printf("\nThe numbers you entered are:\n");
   printnum(num,n);
   sort(num,n);
   printf("\nThe sorted numbers are:\n");
   printnum(num,n);
   getch();
}


/* Function to read a set of numbers */
void readnum(int *a, int n)
{  int i;
   for(i=0;i<n;++i)
   {  printf("Enter Number - %d : ",i+1);
      scanf("%d",a+i);
   }
   return;
}


/* Function to display a set of numbers */
```

```
void printnum(int *a, int n)
{  int i;
   for(i=0;i<n;++i)
      printf("%d\t",*(a+i));
   return;
}


/* Function to sort a set of numbers using selection sort */
void sort(int *a, int n)
{  int i,j,temp;
   for(i=0;i<n-1;++i)
      for(j=i+1;j<n;++j)
         if(*(a+i)>*(a+j))
         {  temp = *(a+i);
            *(a+i)=*(a+j);
            *(a+j)=temp;
         }
   return;
}
```

The following example illustrates the subtraction of two matrices using dynamic memory allocation method.

```
/* Matrix subtraction */
#include<stdio.h>
#include<stdlib.h>
void readmat(int (*x)[10], int r, int c);
void printmat(int (*x)[10], int r, int c);
void submat(int (*x)[10],int (*y)[10], int (*z)[10],int r, int c);
main()
{     int (*mat1)[10],(*mat2)[10],(*mat3)[10],i,j,row,col;
      clrscr();
      printf("\tEnter the Order of the matrices\n");
      printf("\n\n\t\tNumber of rows ? ");
      scanf("%d",&row);
      printf("\t\tNumber of columns ? ");
      scanf("%d",&col);
      mat1= (int *) malloc(row*10*sizeof (int));
      mat2= (int *) malloc(row*10*sizeof (int));
      mat3= (int *) malloc(row*10*sizeof (int));
      clrscr();
      printf("Enter the elements of the first matrix :\n");
      readmat(mat1,row,col);
      printf("Enter the elements of the second matrix :\n");
```

```
readmat(mat2,row,col);
clrscr();
printf("\tFirst Matrix\n");
printf("\t------------\n");
printmat(mat1,row,col);
printf("\n\tSecond Matrix\n");
printf("\t-------------\n");
printmat(mat2,row,col);
submat(mat1,mat2,mat3,row,col);
printf("\n\tDifference Matrix\n");
printf("\t--------------\n");
printmat(mat3,row,col);
getch();
}

/* Function for reading matrix */
void readmat(int (*x)[10],int r,int c)
{     int i,j;
      for(i=0;i<r;++i)
            for(j=0;j<c;++j)
            {     printf("Element (%d,%d) ? ",i+1,j+1);
                  scanf("%d",*(x+i)+j);
            }
      return;
}

/* Function for displaying matrix */
void printmat(int (*x)[10],int r,int c)
{     int i,j;
      for(i=0;i<r;++i)
      {     for(j=0;j<c;++j)
            printf("%6d",*(*(x+i)+j));
            printf("\n");
      }
      return;
}

/* Function for matrix subtraction */
void submat(int (*x)[10],int (*y)[10],int (*z)[10],int r,int c)
{      int i,j;
      for(i=0;i<r;++i)
            for(j=0;j<c;++j)
                  *(*(z+i)+j)=*(*(x+i)+j)-*(*(y+i)+j);
```

```
    return;
}
```

6.14. Advantages of Pointers

- Pointers can be used to pass information back and forth between a function and its reference point.
- Pointers provide a way to return multiple data items from a function via function arguments
- Pointers permits passing function as arguments to another function.
- Pointers provide an alternate way to access individual array elements.
- Pointers provide a convenient way to represent multi-dimensional arrays.

6.15. Structures

A structure in C is a heterogeneous data type. A structure may contain different data types. It groups variables into a single entity. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as members.

6.16. Defining a Structure

A structure definition forms a template that can be used to create structure objects. A structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defied as

```
struct tag {
        member1;
        member2;
        .................
        membern;
          };
```

In this declaration, struct is required keyword; tag is a name that identifies structures having this composition; and member1, member2, ..., membern are individual member declarations. (There is no formal distinction between a structure definition and a structure declaration; the terms are used interchangeably).

The individual members can be ordinary variables, pointers, arrays, or other structures. The member name within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside the structure. Storage-class cannot be assigned to individual members and individual members cannot be initialised within a structure type declaration.

Once the composition of 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 storage 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 below.

```
struct account {
        int acct_no
        char acct_type;
        char name[25];
        float balance;
          };
```

This structure is named account (i.e., the tag is account). It contains four members: an integer quantity (acct_no), a single character (acct_type), a 25-element character array (name[25]), and a floating-point quantity (balance).

We can now declare the structure variables oldcustomer and newcustomer as follows.

```
struct account oldcustomer, newcustomer;
```

Thus, the oldcustomer and newcustomer are structure variables of type account.
It is possible to combine the declaration of structure composition with that of the structure variables, as shown below.

```
storage-class struct tag {
        member1;
        member2;
        .................
        membern;
          } var1, var2, ..., varn;
```

The tag is optional in this situation.

The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct account {
        int acct_no;
        char acct_type;
```

```
    char name[25];
    float balance;
     } oldcustomer, newcustomer;
```

Thus, the oldcustomer and newcustomer are structure variables of type account.

Since the variable declarations are now combined with the declaration of the structure type, the tag (i.e., account) need not be included. Thus, the above declaration can also be written as

```
struct  {
    int acct_no;
    char acct_type;
    char name[25];
    float balance;
     } oldcustomer, newcustomer;
```

A structure variable 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, a C program contains the following structure declaration.

```
struct date {
    int month;
    int day;
    int  year;
      }
struct {
    int acct_no;
    char acct_type;
    char name[25];
    float balance;
    struct date lastpayment;
     } oldcustomer, newcustomer;
```

The second structure account now contains another structure date as one of its members. Note that the declaration of date precedes the declaration of the account.

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial 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 variable = { val1, val2, ..., valn};
```

where val1 refers to the value of the first member, val2 refers to the value of the second member and so on. The following example illustrates the assignment of initial values to the members of a structure variable.

```
struct date {
        int month;
        int day;
        int  year;
          }
    struct account {
        int acct_no;
        char acct_type;
        char name[25];
        float balance;
        struct date lastpayment;
         };
struct account customer = {12345, 'R', "James", 5687.00, 5, 23, 2002};
```

Thus, customer is structure variable of type account, whose members are assigned initial values. The first member (acct_no) is assigned the integer value 12345, the second member (acct_type) is assigned the character 'R', the third member (name[25]) is assigned the string "Jones", and the fourth member (balance) is assigned floating-point value 5687.00. The last member is itself a structure that contains three integer members (month, day and year). Therefore, the last member of customer is assigned the integer values 5, 23 and 2002, respectively.

It is also possible to define an array of structures; i.e., an array in which each element is a structure. The procedure is illustrated in the following example.

```
struct date {
        int month;
        int day;
        int  year;
          }
    struct account {
        int acct_no;
        char acct_type;
        char name[25];
        float balance;
        struct date lastpayment;
         } customer[100];
```

In this declaration customer is a 100-element array of structures. Hence, each element of customer is a separate structure of type account. An array of structures can be assigned initial values just as any other array.

6.17. Processing a Structure

The members of a structure are usually processed individually, as separate entities. A structure member can be accessed by writing

variable.member

where variable refers to the name of the structure-type variable, and member refers to the name of the member within the structure. The period (.) operator separates the variable name form the member name. The period operator is a member of the highest precedence group, and its associativity is left to right. For example, consider the following structure declaration.

```
struct date {
        int month;
        int day;
        int  year;
          }
    struct account {
        int acct_no;
        char acct_type;
        char name[25];
        float balance;
        struct date lastpayment;
         } account;
```

In this example customer is a structure variable of type account. To access the member acct_no, we would write

customer.acct_no

Similarly, customer's name and customer's balance can be accessed by writing

customer.name

and

customer.balance

Since the period operator is a member of the highest precedence group, this operator will take precedence over unary operators as well as various arithmetic, logical and assignment operators. Thus, an expression of the form ++variable.member is equivalent to ++(variable.member); i.e., the ++ operator will apply to the structure member, not the entire structure variable. Similarly, the expression &variable.member is equivalent to &(variable.member); thus, the expression accesses the address of the structure member, not the starting address of the structure variable.

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing

variable.member.submember

where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure. Similarly if a structure member is an array, then the individual array element can be accessed

variable.member[exp]

where exp is a nonnegative value that indicates the array element.
The use period operator can be extended to array of structures, by writing

array[exp].member

where array refers to the array name, and array[exp] is an individual array element. Therefore, array[exp].member will refer to a specific member within a particular structure.

The following program illustrates the use of structures for preparing the result of an examination. This program will read the name and marks in 5 subjects for a set of students and will display their result.

```
/* To prepare the result of an examination */
#include<stdio.h>
main()
{  int i,j,n,sum;
   struct{
         char name[20];
         int mark[5];
         int tot;
         float avg;
         } stud[50];
   clrscr();
   printf("How many students ? ");
   scanf("%d", &n);
   for(i=0;i<n;++i)
   {  clrscr();
      printf("Enter the details of Student-%d\n\n",i+1);
      printf("Name ? ");
      scanf("   %[^\n]",stud[i].name);
      for(sum=0,j=0;j<5;++j)
      {  printf("Mark in Paper-%d ? ",j+1);
         scanf("%d",&stud[i].mark[j]);
         sum+=stud[i].mark[j];
      }
      stud[i].tot=sum;
      stud[i].avg=(float)sum/5.0;
   }
   clrscr();
```

```
   printf("Name\t\tM1\tM2\tM3\tm4\tM5\tTotal\tAvg\tRes\n\n");
   for(i=0;i<n;++i)
   {  printf("%-10s",stud[i].name);
      for(j=0;j<5;++j)
         printf("%8d",stud[i].mark[j]);
      printf("%8d%8.2f",stud[i].tot,stud[i].avg);
      if(stud[i].avg>35.0)
         printf("  Passed\n");
      else
         printf("  Failed\n");
   }
   getch();
}
```

6.18 User Defined Data Types (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, the new variables, arrays, structures, etc. can be declared in terms of this new data type. In general terms, a new data type is defined as

```
typedef type new-type;
```

where type refers to an existing data type (either a standard data type or previous user-defined data type), and new-type refers to the new user-defined data type. The new data type will be new in name only. In reality, this new data type will not be fundamentally different from one of the standard data type. For example, consider the following declaration involving the use of typedef.

```
typedef int age;
```

In this declaration age is a user-defined data type, which is equivalent to type int. Hence, the variable declaration

```
age male, female;
```

is equivalent to writing

```
int male, female.
```

In other words, male and female are regarded as variable of type age, though they are actually integer-type variable.
Similarly, the declarations

```
typedef float height[100];
height men, women
```

define height as 100-element, floating-point array type. Hence, men and women are 100-element floating-point arrays. Another way to express this is

```
typedef height;
height men[100],women[100];
```

The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. Hence, the structure can be referenced more concisely. In general terms, a user defined structure type can be written as

```
typedef struct{
        member1;
        member2;
        .....
        membern;
         } new-type;
```

where new-type is user-defied structure type. Structure variables can then be defined in terms of the new data type.

6.19. Structures and Pointers

The beginning address of a structure can be accessed in the same manner as any other address, through the use of address (&) operator. Thus, if var represents a structure-type variable, then &var represents the starting address of that variable. We can also declare a pointer variable for a structure by writing

```
type *ptrvar;
```

where type is data type that identifies the composition of the structure, and ptrvar represents the name of the pointer variable. The beginning address of structure variable can be assigned to a structure-type pointer variable by writing

```
ptrvar = &var;
```

For example, consider the following structure definition.

```
typedef struct {
        int acct_no;
        char acct_type;
        char name[25];
        float balance;
         } account;
account customer, *pcustomer;
```

In this example, customer is a structure variable of type account, and pcustomer is pointer variable whose object is a structure variable of type

account. Thus, the beginning address of customer can be assigned to the pcustomer by writing

```
pcustomer = & customer;
```

The variable and pointer declarations can be combined with the structure declaration by writing

```
struct{
        member1;
        member2;
        .....
        membern;
         } variable, *ptrvar;
```

The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct {
        int acct_no;
        char acct_type;
        char name[25];
        float balance;
         } customer, *pcustomer;
```

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

```
ptrvar–>member
```

where ptrvar refers to a structure-type pointer variable and –> is comparable to the period (.) operator. Thus the expression

```
ptrvar–>member
```

is equivalent to writing

```
var.member
```

where var is a structure-type variable. The operator –> falls into the highest precedence group. Its associativity is left to right.
The –> operator can be combined with the period operator to access a submember within a structure. Hence, the submember can be accessed by writing

```
ptrvar–>member.submember
```

Similarly, the –> operator can be used to access an element of an array that is a member of the structure. This is accomplished by writing

```
ptrvar–>member[exp]
```

where exp is a nonnegative integer that indicates the array element.
The following program illustrates the use of pointers for processing structures.

```
/* To prepare the result of an examination */
#include<stdio.h>
main()
{  int i,j,n,sum;
   struct{
         char name[20];
         int mark[5];
         int tot;
         float avg;
         } stud[50],*pstud=stud;
   clrscr();
   printf("How many students ? ");
   scanf("%d", &n);
   for(i=0;i<n;++i)
   {  clrscr();
      printf("Enter the details of Student-%d\n\n",i+1);
      printf("Name ? ");
      scanf("   %[^\n]",(pstud+i)–>name);
      for(sum=0,j=0;j<5;++j)
      {  printf("Mark in Paper-%d ? ",j+1);
         scanf("%d",&(pstud+i)–>mark[j]);
         sum+=(pstud+i)–>mark[j];
      }
      (pstud+i)–>tot=sum;
      (pstud+i)–>avg=(float)sum/5.0;
   }
   clrscr();
   printf("Name\t\tM1\tM2\tM3\tm4\tM5\tTotal\tAvg\tRes\n\n");
   for(i=0;i<n;++i)
   {  printf("%-10s",(pstud+i)–>name);
      for(j=0;j<5;++j)
         printf("%8d",(pstud+i)–>mark[j]);
      printf("%8d%8.2f", (pstud+i)–>tot, (pstud+i)–>avg);
      if((pstud+i)–>avg>35.0)
         printf("  Passed\n");
      else
         printf("  Failed\n");
   }
   getch();
}
```

6.20. Unions

A union is a data structure in C that allows the overlay of more than one variable in the same memory area. Unions, like structures, contain members whose individual data types may differ form one another. However, the members within the union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at a time.

In terms of declaration syntax, union is similar to a structure. The only change in declaration is the substitution of the keyword union for the keyword struct. In general terms, the composition of a union is defined as

```
union tag {
        member1;
        member2;
        .................
        membern;
          };
```

where union is required keyword; tag is a name that identifies unions having this composition; and member1, member2, ..., membern are individual member declarations

All members inside a union share storage space. The compiler will allocate sufficient storage for the union members to accommodate the largest member in the union. Other members of the union use the same space. This is how union differs from structure. Individual members in a union occupy the same location in memory. Thus, writing into one will overwrite the other.

Once the composition of union has been defined, individual union-type variables can be declared as follows:

```
storage-class union tag var1, var2, ..., varn;
```

where storage-class is an optional storage class specifier, union is a required keyword, tag is the name that appeared in the union declaration, and var1, var2, ..., varn are union variables of type tag.

A typical union declaration is shown below.

```
union id {
        char colour[12];
        int size;
         };
```

This union is named id (i.e., the tag is id). It contains two members: a 12-element character array (colour[12]) and an integer quantity (size). The 12-element character string will require more storage area within the computer's memory than integer quantity. Therefore, a block of memory large enough for the 12-element character string will be allocated to each

union variable. The compiler will automatically distinguish between the 12-element character array and the integer quantity within the given block of memory, as required.

We can now declare the union variables shirt and blouse as follows.

union id shirt, blouse;

Thus, the shirt and blouse are union variables of type id. Each variable can represent either a 12-element character string (colour) or an integer quantity (size) at any one time.

It is possible to combine the declaration of union composition with that of the union variables, as shown below.

```
storage-class union tag {
        member1;
        member2;
        .............
        membern;
          } var1, var2, ..., varn;
```

The tag is optional in this situation.

The following single declaration is equivalent to the two declarations presented in the previous example.

```
union id {
        char colour[12];
        int size;
         }shirt, blouse;
```

Thus, the shirt and blouse are union variables of type id.

Since the variable declarations are now combined with the declaration of the union type, the tag (i.e., id) need not be included. Thus, the above declaration can also be written as

```
union {
        char colour[12];
        int size;
         }shirt, blouse;
```

A union may be a member of a structure, and a structure may be member of union.

For example, consider the following declarations.

```
union id {
        char colour[12];
        int size;
        };
struct clothes {
        char manufacturer[20];
        float cost;
```

```
        union id description;
        } shirt, blouse;
```

Now shirt and blouse are structures variable of type clothes. Each variable will contain the following members: a string (manufacturer) , a floating-point quantity (cost), and a union (description). The union may represent either a string (colour) or an integer quantity.

An individual union member can be accessed in the same manner as an individual structure member, using the operators **.** and **–>**. Thus, if var is a union variable, then var.member refers to a member of the union. Similarly, if ptrvar is pointer variable that points to a union, then ptrvar–>member refers to a member of that union.

Questions

3 Mark Questions

1. What are arrays? How are arrays defined?
2. In what way does an array differ from an ordinary variable?
3. What are subscripts? What restrictions apply to the values that can be assigned to subscripts?
4. What does the name of an array signify?
5. How are initial values assigned to array elements?
6. How are strings stored in an array in C?
7. How are arrays passed to function?
8. When passing an argument to a function, what is the difference between passing by value and passing by reference?
9. What is the function of a pointer variable? What are its uses?
10. Explain the functions of the address operator (&) and indirection (*) operator.
11. How is a pointer variable declared? What is the purpose of the data type included in the declaration?
12. What are the operations that can be carried out over pointers? Explain.
13. What is the relationship between an array name and a pointer?
14. Describe two different ways to specify the address of an array element.
15. Describe two different ways to access an array element.
16. How are arrays defined in terms of pointer variables? How does it differ from an ordinary array definition?
17. What is meant by dynamic memory allocation? How is it accomplished in C?
18. What is the purpose of the library functions malloc and calloc? Hoe do they differ?
19. What is a structure? How is a structure defined?
20. What is a structure? How does a structure differ from an array?
21. Explain the function of the period operator? What is its precedence and associativity?
22. What is a union? How does it differ from a structure?
23. What is union? How is a union defined?

Programming Questions

Draw flowcharts and write C programs for the following

1. To find the largest and smallest among a set of numbers stored in an array. Display the set of numbers along with the smallest and largest.
2. To find the largest odd and even numbers from a set of numbers stored an array. Display the set of numbers along with the largest odd and even numbers.
3. To delete the duplicate numbers from a set of numbers stored in array. Display the set of numbers before and after deletion.
4. To display the separate sum and average of positive and negative numbers stored in an array. Display the set of numbers along with the sum and averages.
5. To find the smallest and largest differences between any two numbers in a set of numbers stored in an array. Display the respective elements and their location details.
6. To search a number in a set of numbers stored in an array. Display the set of numbers, the element searched and its location in the array.
7. To sort a set of numbers in ascending order.
8. To sort a set of numbers in descending order.
9. To insert a number into a set of numbers stored in an array. Display the set of numbers before and after insertion.
10. To delete a number from a set of numbers stored in an array. Display the set of numbers before and after deletion.
11. To merge two sets of numbers stored in to separate arrays. For this sort the individual set of numbers. Then add the elements form the individual arrays to a third array so that the resultant is a sorted array.
12. To append one set of numbers to another set of numbers stored in an array. Display the individual sets of numbers along with the appended array.
13. To input 'n' pairs of coordinates of points and display whether each point belongs to first, second, third or fourth quadrant, origin, on –ve side of X-axis, on +ve side of X-axis, on –ve side of Y-axis or +ve side of Y-axis.
14. To display the count and percentage of positive numbers, negative numbers and zeroes in a set of numbers stored in an array. Display the set of numbers along with the result.
15. To read a given 4-digit number in words.
16. To find the standard deviation of a set of numbers using $\sqrt{(d_1^2+ d_2^2+ d_3^2+.....+ d_n^2)/n}$ where $d_i=(x_i-\bar{x})$, i = 1,2,3,....,n and $\bar{x}$ is the mean.

17. To display the sum of the individual elements of an m×n matrix. Display the matrix and the sum.
18. To display the sum of individual rows of an m×n matrix. Display the matrix and the results.
19. To display the sum of individual columns of an m×n matrix. Display the matrix and the results.
20. To display the sum of diagonal elements of a square matrix. Display the matrix and the result.
21. To display the sum of all non-diagonal elements of a square matrix. Display the matrix and the result.
22. To display the sum of individual rows and columns of an m×n matrix. Display the matrix and the results.
23. To sum the individual elements of upper and lower triangular matrices of a square matrix. Display the square matrix, triangular matrices and their sum.
24. To find the sum of two m×n matrices. Display the matrices along with the sum matrix.
25. To find the difference of two m×n matrices. Display the matrices along with the difference matrix.
26. To find the product of two matrices. Display the matrices along with the product matrix.
27. To sort the individual rows of an m×n matrix. Display the matrix before and after sorting.
28. To sort the individual columns of an m×n matrix. Display the matrix before and after sorting.
29. To find the biggest and smallest elements of an m×n matrix. Display the matrix along with the results.
30. To compute the norm of a matrix. Norm of a matrix is the square root of the sum of the squares of individual elements. Display the matrix along with the result.
31. To check whether the given square matrix is a magic square. A matrix is a magic square if Column sums = Row sums = Diagonal sum and the element in the matrix should be distinct. Check your program by entering the elements 4 3 8, 9 5 1, and 2 7 6 in row-wise.
32. To check whether a given matrix is orthogonal. A matrix M is orthogonal if $M \times M^T$= Identity matrix (A matrix with diagonal elements are 1s and all other elements are 0s). Display M, M^T, $M \times M^T$ and the result.
33. To find the transpose of a m×n matrix. Display the matrix and its transpose.
34. To find the inverse of a matrix. Display the matrix and its inverse.
35. To display the first 'n' lines of the Pascal's triangle. A Pascal's is shown below

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

36. To convert Roman numerals into decimal.
37. To convert decimal to binary.
38. To convert decimal to octal.
39. To convert decimal to hexadecimal.
40. To convert a hexadecimal number to decimal.
41. To convert an octal number to decimal.
42. To convert a binary number to decimal.
43. To covert a number form any base to any other base, for a base that ranges from 1 to 16.
44. To input as string and to output the frequency of occurrences of each alphabet A through Z, by ignoring the case.
45. To input a string and to display the length of the string in terms of the number of characters in the string. Ignore the null character at the end of the string. Use a function which, given a string argument ‘s1’, returns the length of the string s1. Do not use any string library function for this purpose.
46. To compare two string and to display whether a string is less than, grater than or equal to the other. Use a function which, given two string arguments ‘s1’ and ‘s2’, returns a positive value if s1>s2, returns zero if s1=s2, and returns a negative value if s1<s2. The comparison is to be performed on a character-by-character basis using the collating sequence of the underlying machine. Do not use any string library function for this purpose.
47. To compare two string and to display whether a string is less than, grater than or equal to the other, by ignoring the case. Use a function which, given two string arguments ‘s1’ and ‘s2’, returns a positive value if s1>s2, returns zero if s1=s2, and returns a negative value if s1<s2. The comparison is to be performed on a character-by-character basis using the collating sequence of the underlying machine. Do not use any string library function for this purpose.
48. To check whether a string occurs in a given string. Use a function which, given two string arguments ‘s1’ and ‘s2’, searches s1 to find the occurrence of the string s2 in s1. If s2 is found, the value returned is the positron of in the s1; otherwise, the value returned is negative. Do not use any string library function for this purpose.
49. To check whether a string occurs in a given string, by ignoring the case. Use a function which, given two string arguments ‘s1’ and ‘s2’, searches s1 to find the occurrence of the string s2 in s1. If s2 is found,

the value returned is the position of s2 in the s1; otherwise, the value returned is negative. Do not use any string library function for this purpose.

50. To delete some characters from a string. Use a function which, given a string argument ‘s1’ and two integer arguments ‘n’ and ‘amount’, delete amount characters from s1 starting at position n. Do not use any string library function for this purpose.
51. To convert a string into all uppercase. Use a function which, given a string argument ‘s1’, converts all lowercase letters to uppercase, leaving the others changed. Do not use any string library function for this purpose.
52. To convert a string into all lowercase. Use a function which, given a string argument ‘s1’, converts all uppercase letters to lowercase, leaving the others changed. Do not use any string library function for this purpose.
53. To input a string and to determine whether any characters are repeating in the string. Use a function which, given a string argument ‘s1’, returns true if all characters are distinct and false if any character is repeated. Do not use string library function for this purpose.
54. To insert a string into another. Use a function which, given two string arguments ‘s1’ and ‘s2’ and an integer argument ‘n’, inserts s2 into s1, starting from n. Do not use string library function for this purpose.
55. To concatenate two strings. Use a function which, given two string arguments ‘s1’ and ‘s2’, concatenates s2 to s1. Do not use any string library function for this purpose.
56. To reverse a string. Use a function which, given a string argument ‘s1’, reverses s1. Do not use any string library function for this purpose.
57. To input two strings and to check whether one is anagram of the other, that is, whether one string is a permutation of characters in the other string, e.g., mate and team. Use a function which, given two string arguments ‘s1’ and ‘s2’, returns 1 if one is anagram of the other; 0 otherwise.
58. To input a set of strings and to display the longest string and its length. (You may use suitable string library functions from this question onwards)
59. To input a set of strings and display their lengths in a tabular form. Also, display the total and average number of characters in the strings.
60. To input a set of names and to sort it in ascending order. Display the names before and after sorting.
61. To input a set of names and to sort it in descending order. Display the names before and after sorting.
62. To count and display the number of words in a string.
63. To display the words of a string in ascending order.

64. To display the words of a string in descending order.
65. To display the characters in a string in ascending order.
66. To display the characters in a string in descending order.
67. To input a string and to display the ASCII equivalent of each character in a tabular form.
68. To count the number of occurrences of each vowels in a string. Display the count of each vowel separately. Also, display the total number of vowels.
69. To count and display the number of vowels, consonants, digits and other characters in a string.
70. To check whether a string is palindrome or not.
71. To read a line of text and to squeeze out all blanks from it and output the line of text with no blanks.
72. To encrypt a string using the strategy of replacing a letter by the next letter in its collating sequence. Thus every A will be replaced by B, every B by C and so on and finally Z will be replaced by A. Blanks are left undisturbed.

Answers to Select Questions

(Flowcharting is left as an exercise to the students)

2.

```
/* To find and display largest Even and Odd */
#include<stdio.h>
main()
{  int i,n,num[50],oddlarge,evenlarge,evenflag=0,oddflag=0;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   if(n<=0)
      printf("Invalid data");
      else
      {
         for(i=0;i<n;i++)
         {  printf("Enter Number-%d : ", i+1);
            scanf("%d", &num[i]);
         }
         for(i=0;i<n;i++)
         {     if(num[i]%2 == 0)
               {     if(!(evenflag))
                     {     evenlarge=num[i];
                           evenflag=1;
                     }
                     else
                        if(evenlarge< num[i])
                              evenlarge = num[i];
               }
               else
               {  if(!(oddflag))
                     {     oddlarge=num[i];
                           oddflag=1;
                     }
                     else
                        if(oddlarge<num[i])
                              oddlarge=num[i];
               }
         }
         clrscr();
         printf("\nThe numbers you entered are :\n");
         for(i=0;i<n;i++)
         printf("%d\t",num[i]);
```

```
        if(evenflag)
          printf("\nThe largest Even number is %d\n",evenlarge);
        else
          printf("\nNo Even numbers in your set of numbers\n");
        if(oddflag)
          printf("The largest Odd number is %d\n",oddlarge);
        else
          printf("No Odd numbers in your set of numbers\n");
      }
      getch();
}
```

4.

```
/* To calculate and display sum and average of +ve and -ve numbers */
#include<stdio.h>
main()
{  int i,n,poscount=0,negcount=0;
   float num[50],negsum=0.0,possum=0.0,negavg,posavg;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   if(n<=0)
     printf(“Invalid data”);
   else
   {
     for(i=0;i<n;i++)
     {  printf("Enter Number-%d : ", i+1);
        scanf("%f", &num[i]);
        if(num[i] < 0)
        {  negsum+=num[i];
        negcount++;
        }
        if(num[i] > 0)
        {  possum+=num[i];
           poscount++;
        }
     }
     clrscr();
     printf("\nThe numbers you entered are :\n");
     for(i=0;i<n;i++)
        printf("%g\t",num[i]);
     if(negcount)
     {  negavg = negsum/negcount;
        printf("\nSum of negative numbers is %g",negsum);
```

```
        printf("\nAverage of negative numbers is %g",negavg);
      }
      else
        printf("\nNo negative numbers exist in your set of numbers");
      if(poscount)
      {  posavg = possum/poscount;
        printf("\nSum of positive numbers is %g",possum);
        printf("\nAverage of positive numbers is %g",posavg);
      }
      else
        printf("\nNo positive numbers exist in your set of numbers");
   }
   getch();
}
```

5.

```
/* To calculate and display smallest and largest difference */
#include<stdio.h>
main()
{  int i,j,n;
   float num[50],temp;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   if(n<=0)
      printf(“Invalid data”);
   else
   {
      for(i=0;i<n;i++)
      {  printf("Enter Number-%d : ", i+1);
        scanf("%f", &num[i]);
      }
      clrscr();
      printf("\nThe numbers you entered are :\n");
      for(i=0;i<n;i++)
        printf("%g \t",num[i]);
      for(i=0;i<n-1;i++)
        for(j=i+1;j<n;++j)
          if(num[i]>num[j])
          {  temp = num[i];
            num[i]=num[j];
            num[j]=temp;
          }
```

```
    printf("\nThe largest difference is %g",num[n-1]-num[0]);
    printf("\nCorresponding elements are %g and %g",num[n-1],num[0]);
    printf("\nThe Smallest difference is %g",num[0]-num[n-1]);
    printf("\nCorresponding elements are %g and %g",num[0],num[n-1]);
  }
  getch();
}
```

6.

```
/* To Search a number in a set of numbers */
#include<stdio.h>
main()
{  int i,n,count=0;
   float num[50],search;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   if(n<=0)
      printf(“Invalid data”);
   else
   {
      for(i=0;i<n;i++)
      {  printf("Enter Number-%d : ", i+1);
         scanf("%f", &num[i]);
      }
      printf("\n\n\nNumber to searched  ? ");
      scanf("%f",&search);
      clrscr();
      printf("\nThe numbers you entered are :\n");
      for(i=0;i<n;i++)
         printf("%g \t",num[i]);
      printf("\n\n\n");
      for(i=0;i<n;i++)
         if(num[i] == search)
         {  printf("Location-%d\n",i);
            count++;
         }
      if(count)
         printf("\nThe number %g exists in %d locations shown above",
                                    search,count);
      else
         printf("\nThe number %g does not exists in the set",search);
   }
```

```
   getch();
}
```

9.

```
/* To insert a number into a set of numbers*/
#include<stdio.h>
main()
{  int i,n,loc;
   float num[50],insert,temp1,temp2;
   clrscr();
   printf("How many numbers ? ");
   scanf("%d",&n);
   if(n<=0)
      printf(“Invalid data”);
   else
   {
      for(i=0;i<n;i++)
      {  printf("Enter Number-%d : ", i+1);
         scanf("%f", &num[i]);
      }
      printf("\n\n\nNumber to inserted  ? ");
      scanf("%f",&insert);
      printf("\n\n\nLocation ? ");
      scanf("%d",&loc);
      if(location<0 || location >n)
         printf(“Invalid location”);
      else
      {
         clrscr();
         printf("\nThe numbers you entered are :\n");
         for(i=0;i<n;i++)
            printf("%g \t",num[i]);
         temp1=num[loc];
         num[loc]=insert;
         for(i=loc+1;i<=n;i++)
         {  temp2=num[i];
            num[i]=temp1;
            temp1=temp2;
         }
         printf("\nThe numbers after insertion :\n");
         for(i=0;i<=n;i++)
            printf("%g \t",num[i]);
      }
```

```
   }
   getch();
}
```

11.

```
/* To merge two array into a third array */
#include<stdio.h>
void sort(float a[], int n);
void display(float a[], int n);
void readnum(float a[], int n);
main()
{  int i,j,k,n1,n2;
   float num1[50],num2[50],num3[100];
   clrscr();
   printf("How many elements in First set ? ");
   scanf("%d",&n1);
   if(n1<=0)
      printf("Invalid data");
   else
   {
      printf("Enter the elements of the First set :\n");
      readnum(num1,n1);
      clrscr();
      printf("How many elements in Second set ? ");
      scanf("%d",&n2);
      if(n2<=0)
         printf("Invalid data");
      else
      {
         printf("Enter the elements of the Second set :\n");
         readnum(num2,n2);
         sort(num1,n1);
         sort(num2,n2);
         for(i=0,j=0,k=0;i<(n1+n2);i++)
         {  if((j<n1)&&(k<n2))
               if(num1[j]<num2[k])
                  num3[i]=num1[j++];
               else
                  num3[i]=num2[k++];
            else
               if(k==n2)
                  num3[i]=num1[j++];
               else
                  if(j==n1)
```

```
                    num3[i]=num2[k++];
            }
            clrscr();
            printf("Your First set is :\n");
            display(num1,n1);
            printf("Your Second set is :\n");
            display(num2,n2);
            printf("The Merged set is :\n");
            display(num3,n1+n2);
        }
    }
    getch();
}

/* Function to sort an array */
void sort(float a[], int n)
{  float temp;
   int i,j;
   for(i=0;i<n-1;i++)
      for(j=i+1;j<n;++j)
         if(a[i]>a[j])
         {  temp=a[i];
            a[i]=a[j];
            a[j]=temp;
         }
   return;
}

/* Function to display an array */
void display(float a[], int n)
{  int i;
   for(i=0;i<n;++i)
      printf("%g\t",a[i]);
   printf("\n\n\n");
   return;
}

/* Function to read an array */
void readnum(float a[], int n)
{  int i;
   for(i=0;i<n;++i)
   {  printf("Number-%d ? " ,i+1);
      scanf("%f",&a[i]);
   }
```

```
   return;
}
```

13.

```
/* To display the positions of the coordinates */
#include<stdio.h>
main()
{  int i,n;
   float x[50],y[50];
   clrscr();
   printf("How many Coordinates ? ");
   scanf("%d",&n);
   if(n<=0)
      printf("Invalid data");
   else
   {  for(i=0;i<n;i++)
      {  printf("Enter Set - %d\n",i+1);
         printf("\tX-axis value ? ");
         scanf("%f", &x[i]);
         printf("\tY-axis value ? ");
         scanf("%f", &y[i]);
      }
      clrscr();
      printf("\nCoordinates\tPosition");
      printf("\n----------\t---------\n");
      for(i=0;i<n;i++)
      {  printf("\n(%g,%g)\t\t",x[i],y[i]);
         if((x[i]==0)&&(y[i]==0))
            printf("Origin");
         else
            if((x[i]==0)&&(y[i]>0))
               printf("+Ve Side of Y-axis");
            else
               if((x[i]==0)&&(y[i]<0))
                  printf("-Ve Side of Y-axis");
               else
                  if((x[i]<0)&&(y[i]==0))
                     printf("-Ve Side of X-axis");
                  else
                     if((x[i]>0)&&(y[i]==0))
                        printf("+Ve Side of X-axis");
                     else
                        if((x[i]>0)&&(y[i]>0))
```

```
                    printf("I st Quadrant");
                else
                    if((x[i]<0)&&(y[i]>0))
                        printf("II nd Quadrant");
                    else
                        if((x[i]<0)&&(y[i]<0))
                            printf("III rd Quadrant");
                        else
                            printf("IV th Quadrant");
        }
    printf("\n----------\t---------\n");
    }
    getch();
}
```

16.

```
/* To calculate and display Standard Deviation */
#include<stdio.h>
#include<math.h>
main()
{  int i,n;
    float num[50];
    double var=0.0,sd,mean,sum=0.0;
    clrscr();
    printf("How many numbers ? ");
    scanf("%d",&n);
    if(n<=0)
        printf("Invalid data");
    else
    {  for(i=0;i<n;i++)
        {  printf("Enter Number-%d : ", i+1);
            scanf("%f", &num[i]);
            sum+=num[i];
        }
        mean = sum/n;
        for(i=0;i<n;i++)
            var += var + pow((num[i]-mean),2);
        var = var/n;
        sd=sqrt(var);
        printf("The Numbers You entered are:\n");
        for(i=0;i<n;i++)
            printf("%g\t",num[i]);
        printf("\nMean is %g ",mean);
```

```
      printf("\Variance is %g ",var);
      printf("\Standard Deviation is %g ",sd);
   }
   getch();
}
```

22.

```
/* To find the sum of individual rows and columns of matrix */
#include<stdio.h>
void readmat(int x[][10], int r, int c);
void printmat(int x[][10], int r, int c);
int addrow(int x[][10], int r, int c);
int addcol(int x[][10], int r, int c);
main()
{  int i,mat[10][10],row,col;
   clrscr();
   printf("\tEnter the Order of the matrix\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row);
   printf("\t\tNumber of columns ? ");
   scanf("%d",&col);
   if((row<=0) || (col<=0))
      printf("Invalid Order");
   else
   {
      clrscr();
      printf("Enter the elements of the Matrix :\n");
      readmat(mat,row,col);
      clrscr();
      printf("\tYour Matrix\n");
      printf("\t------------\n");
      printmat(mat,row,col);
      for(i=0;i<row;++i)
         printf("\nSum of row - %d is %d",i+1,addrow(mat,i,col));
      for(i=0;i<col;++i)
         printf("\nSum of column - %d is %d",i+1,addcol(mat,row,i));
    }
  getch();
}

/* Function for reading matrix */
void readmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
```

```
      for(j=0;j<c;++j)
      {  printf("Element (%d,%d) ? ",i+1,j+1);
         scanf("%d",&x[i][j]);
      }
   return;
}

/* Function for displaying matrix */
void printmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<c;++j)
         printf("%6d",x[i][j]);
      printf("\n");
   }
   return;
}

/* Function for adding rows */
int addrow(int x[][10],int r, int c)
{  int i,sum=0;
   for(i=0;i<c;++i)
            sum = sum + x[r][i];
   return;
}

/* Function for adding columns */
int addcol(int x[][10],int r, int c)
{  int i,sum=0;
   for(i=0;i<r;++i)
            sum = sum + x[i][c];
   return;
}
```

27.

```
/* To sort individual rows of a matrix */
#include<stdio.h>
void readmat(int x[][10], int r, int c);
void printmat(int x[][10], int r, int c);
void sortrow(int x[][10], int r, int c);
main()
{  int i,mat[10][10],row,col;
   clrscr();
```

```
   printf("\tEnter the Order of the matrix\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row);
   printf("\t\tNumber of columns ? ");
   scanf("%d",&col);
   if((row<=0) || (col<=0))
      printf("Invalid Order");
   else
   {
      clrscr();
      printf("Enter the elements of the Matrix :\n");
      readmat(mat,row,col);
      clrscr();
      printf("\tYour Matrix\n");
      printf("\t------------\n");
      printmat(mat,row,col);
      for(i=0;i<row;++i)
         sortrow(mat,i,col);
      printf("\tMatrix after sorting\n");
      printf("\t--------------------\n");
      printmat(mat,row,col);
   }
  getch();
}

/* Function for reading matrix */
void readmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
      {  printf("Element (%d,%d) ? ",i+1,j+1);
         scanf("%d",&x[i][j]);
      }
   return;
}
/* Function for displaying matrix */
void printmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<c;++j)
         printf("%6d",x[i][j]);
      printf("\n");
   }
   return;
}
```

```
/* Function for sorting rows */
void sortrow(int x[][10],int r, int c)
{  int i,j,temp;
   for(i=0;i<c-1;i++)
      for(j=i+1;j<c;++j)
         if(x[r][i]>x[r][j])
         {  temp = x[r][i];
            x[r][i]=x[r][j];
            x[r][j]=temp;
         }
return;
}
```

33.

```
/* To find the Transpose of matrix */
#include<stdio.h>
void readmat(int x[][10], int r, int c);
void printmat(int x[][10], int r, int c);
void transpose(int x[][10],int y[][10],int r,int c);
main()
{  int mat[10][10],tra[10][10],row,col;
   clrscr();
   printf("\tEnter the Order of the matrix\n");
   printf("\n\n\t\tNumber of rows ? ");
   scanf("%d",&row);
   printf("\t\tNumber of columns ? ");
   scanf("%d",&col);
   if((row<=0) || (col<=0))
      printf("Invalid Order");
   else
   {
      clrscr();
      printf("Enter the elements of the Matrix :\n");
      readmat(mat,row,col);
      clrscr();
      printf("\tYour Matrix\n");
      printf("\t------------\n");
      printmat(mat,row,col);
      transpose(mat,tra,row,col);
      printf("\tTranspose of the Matrix\n");
      printf("\t---------------------\n");
      printmat(tra,col,row);
    }
```

```
  getch();
}

/* Function for reading matrix */
void readmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
      for(j=0;j<c;++j)
      {  printf("Element (%d,%d) ? ",i+1,j+1);
         scanf("%d",&x[i][j]);
      }
   return;
}

/* Function for displaying matrix */
void printmat(int x[][10],int r,int c)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<c;++j)
         printf("%6d",x[i][j]);
      printf("\n");
   }
   return;
}

void transpose(int x[][10],int y[][10],int r,int c)
{  int i,j;
   for(i=0;i<c;++i)
      for(j=0;j<r;++j)
         y[i][j]=x[j][i];
   return;
}
```

35.

```
/* To generate the Pascal's Triangle */
#include<stdio.h>
void generate(int x[][100], int r);
void print(int x[][100], int r);
main()
{  int tria[100][100],n;
   clrscr();
   printf("How many lines ? ");
   scanf("%d",&n);
   if(n<=0)
```

```
      printf("Invalid Data");
   else
   {  clrscr();
      generate(tria,n);
      printf("\tFirst %d lines of Pascal's Triangle\n",n);
      print(tria,n);
    }
  getch();
}


/* Function for generating the Pascal's triangle */
void generate(int x[][100],int r)
{  int i,j;
   for(i=0;i<r;++i)
   {  x[i][0]=1;
      for(j=1;j<=i-1;++j)
         x[i][j]=x[i-1][j]+x[i-1][j-1];
      x[i][j]=1;
   }
   return;
}

/* Function for displaying the Pascal's triangle */
void print(int x[][100],int r)
{  int i,j;
   for(i=0;i<r;++i)
   {  for(j=0;j<=i;++j)
         printf("%6d",x[i][j]);
      printf("\n");
   }
   return;
}
```

36.

```
/* To convert the Roman number into equivalent decimal */
#include<stdio.h>
#include<ctype.h>
int value (char dig);
main()
{  char roman[10];
   int val1,val2,decequiv=0,i=0;
   clrscr();
```

```
   printf("Enter the Roman Number : ");
   scanf("%[^\n]",roman);
   val1 = value(roman[i]);
   for(i=1;roman[i] != '\0';++i)
   {  val2 = value(roman[i]);
      if((val1==0) || (val2 ==0))
         {
            printf("Invalid Character in Roman Number");
            getch();
            exit(0);
         }
      if(val1>=val2)
         decequiv+=val1;
      else
         decequiv-=val1;
      val1=val2;
   }
   decequiv+=val1;
   printf("Decimal equivalent of %s is %d",roman,decequiv);
  getch();
}

/* Function for generating value of roman digit */
int value(char dig)
{  switch(toupper(dig))
   {  case 'M': return(1000);
                break;
      case 'C': return(100);
                break;
      case 'L': return(50);
                break;
      case 'X': return(10);
                break;
      case 'V': return(5);
                break;
      case 'I':  return(1);
                break;
      default : return(0);
   }
}
```

43.

```
/*Number conversion */
#include<stdio.h>
```

```
#include<ctype.h>
#include<string.h>
#include<math.h>
int value(char dig);
char letter(int  dig);
main()
{
   int oldbase,newbase,len,i,j,d,val;
   long dec=0;
   char oldnum[100],newnum[100];
   clrscr();
   printf("What is the Base of the number to be converted ? ");
   scanf("%d",&oldbase);
   printf("Enter Your Number : ");
   scanf("  %[^\n]",oldnum);
   printf("What is the New Base ? ");
   scanf("%d",&newbase);
   len=strlen(oldnum);
   for(i=len-1,j=0;i>=0;--i,j++)
   { val=value(oldnum[i]);
     if((val>(oldbase-1)) || (val == -1))
     {   printf("Invalid Character in Your Number ");
         getch();
         exit(0);
     }
     dec = dec + pow(oldbase,j)*val;
   }
   for(i=0;dec>0;++i)
   {  d = dec % newbase;
      newnum[i]=letter(d);
      dec = dec/newbase;
   }
   newnum[i]='\0';
   printf("Number %s in Base %d is %s in Base
                        %d",oldnum,oldbase,strrev(newnum),newbase);
  getch();
}

/* Function to determine the decimal value of the character */
int value(char dig)
{  switch(toupper(dig))
   {  case '0':  return(0);
                 break;
      case '1':  return(1);
```

```
                break;
      case '2':  return(2);
                break;
      case '3':  return(3);
                break;
      case '4':  return(4);
                break;
      case '5':  return(5);
                break;
      case '6':  return(6);
                break;
      case '7':  return(7);
                break;
      case '8':  return(8);
                break;
      case '9':  return(9);
                break;
      case 'A': return(10);
                break;
      case 'B': return(11);
                break;
      case 'C': return(12);
                break;
      case 'D': return(13);
                break;
      case 'E': return(14);
                break;
      case 'F': return(15);
                break;
      default : return(-1);
   }
}

/* Function for character form of the digit  */
char letter(int  dig)
{  switch(dig)
   {  case 0:   return('0');
                break;
      case 1:   return('1');
                break;
      case 2:   return('2');
                break;
      case 3:   return('3');
                break;
      case 4:   return('4');
```

```
                break;
        case 5:  return('5');
                break;
        case 6:  return('6');
                break;
        case 7:  return('7');
                break;
        case 8:  return('8');
                break;
        case 9:  return('9');
                break;
        case 10: return('A');
                break;
        case 11: return('B');
                break;
        case 12: return('C');
                break;
        case 13: return('D');
                break;
        case 14: return('E');
                break;
        case 15: return('F');
                break;
    }
}
```

44.

```
/* To display the frequency of alphabets A-Z in a text */
#include<stdio.h>
main()
{  char text[80];
   int i,count[26];
   clrscr();
   for(i=0;i<26;++i)
      count[i]=0;
   printf("Enter Your Text : ");
   gets(text);
   for(i=0;text[i] != '\0';++i)
      {  text[i]=(text[i]>='a' && text[i] <='z') ? ('A'+text[i]-'a') : text[i];
         if (text[i]>'A' && text[i]<='Z')
            ++count[text[i]-'A'];
      }
   printf("Letter\tFrequency");
```

```
   printf("\n-------\t--------");
   for(i=0;i<26;++i)
   printf("\n%c\t%d",i+65,count[i]);
   getch();
}
```

45.

```
/* To display the length of a string */
#include<stdio.h>
int lenstr(char s1[]);
main()
{  char text[80];
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   printf("String %s contains %d characters",text,lenstr(text));
   getch();
}
int lenstr(char s1[])
{  int i;
   for(i=0;s1[i] != '\0';++i);
   return i;
}
```

46.

```
/* To compare two strings */
#include<stdio.h>
int cmpstr(char s1[], char s2[]);
main()
{  char text1[80],text2[80];
   clrscr();
   printf("Enter Your First Text : ");
   gets(text1);
   printf("Enter Your Second Text : ");
   gets(text2);
   switch(cmpstr(text1,text2))
   {  case -1: printf("%s is less than %s",text1,text2);
                        break;
      case  0: printf("%s and %s are equal",text1,text2);
                        break;
      case  1: printf("%s is greater than %s",text1,text2);
                        break;
   }
```

```
   getch();
}

int cmpstr(char s1[], char s2[])
{  int i,eqlflag=1;
   for(i=0;s1[i] != '\0'&& s2[i] != '\0';++i)
   {
      if (s1[i]>s2[i]) return 1;
      if(s1[i]!=s2[i])
         eqlflag=0;
   }
   if((eqlflag) && (s1[i]=='\0') && s2[i]=='\0')
      return 0;
   else
      if(s1[i] != '\0')
         return 1;
      else
         return -1;
}
```

49.

```
/* To check a string s2 occurs in string s1 */
#include<stdio.h>
int stringstrig(char s1[], char s2[]);
main()
{  char text1[80],text2[80];
   int loc;
   clrscr();
   printf("Enter Your Text : ");
   gets(text1);
   printf("Enter the String to be searched : ");
   gets(text2);
   loc=stringstrig(text1,text2);
   if(loc>=0)
      printf("\"%s\" occurs in \"%s\" starting from location %d",
                                    text2,text1,loc);
   else
      printf("%s does not occurs in %s",text2,text1);
   getch();
}

int stringstrig(char s1[], char s2[])
```

```
{  int i,j,isflag;
   for(i=0;s1[i] != '\0';++i)
   {  isflag = 1;
      if (s1[i]==s2[0])
      {  for(j=0;s2[j]!='\0';++j)
            if(s1[i+j] != s2[j])
               {  isflag = 0;
                  break;
               }
         if(isflag) return i;
      }
   }
   return -1;
}
```

51.

```
#include<stdio.h>
void uprstr(char s1[]);
main()
{  char text[80];
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   uprstr(text);
   printf("Its uppercase Equivalent is %s",text);
   getch();
}

void uprstr(char s1[])
{  int i;
   for(i=0;s1[i] != '\0';++i)
      s1[i]=(s1[i]>='a' && s1[i] <='z') ? ('A'+s1[i]-'a') : s1[i];
}
```

56.

```
#include<stdio.h>
void revstr(char s1[]);
main()
{  char text[80];
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   printf("Its Reverse is ");
```

```
   revstr(text);
   getch();
}


void revstr(char s1[])
{  int i;
   for(i=0;s1[i] != '\0';++i);
   for(;i>=0;--i)
      putchar(s1[i]);
}
```

62.

```
#include<stdio.h>
#include<string.h>
int wrdcount(char s1[]);
int check(char c);
main()
{  char text[80];
   int len;
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   len = strlen(text);
   text[len]=' ';
   text[len+1]='\0';
   printf("%s contains %d Words",text,wrdcount(text));
   getch();
}

int wrdcount(char s1[])
{  int i,count=0;
   for(i=0;s1[i] != '\0';++i)
   if( (check(s1[i])) && (!check(s1[i+1])) )
      count++;
   return count;
}

int check(char c)
{  switch(c)
   {  case '.' :
      case ';' :
      case ',' :
```

```
        case '\t':
        case ' ' : return 1;
                   break;
        default  : return 0;
    }
}
```

63.

```
/* To sort the individual words of text */
#include<stdio.h>
#include<string.h>
void words(char s1[]);
int check(char c);
main()
{  char text[80];
   int len;
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   len = strlen(text);
   text[len]=' ';
   text[len+1]='\0';
   words(text);
   getch();
}

void words(char s1[])
{  int i,j=0,k=0,n;
   char wrds[50][20],temp[20];
   clrscr();
   for(i=0;s1[i] != '\0';++i)
   {  if(!check(s1[i]))
         wrds[j][k++]=s1[i];
      if( (check(s1[i])) && (!check(s1[i+1])) )
      {  wrds[j][k]='\0';
         j++;
         k=0;
      }
   }
   printf("Words In Your Text Are:\n");
   for(i=0;i<j;++i)
      printf("%s\n",wrds[i]);
   n=j;
   for(i=0;i<n-1;++i)
```

```
      for(j=0;j<(n-1)-i;++j)
         if(strcmp(wrds[j],wrds[j+1]) > 0)
         {  strcpy(temp,wrds[j]);
            strcpy(wrds[j],wrds[j+1]);
            strcpy(wrds[j+1],temp);
         }
   printf("\n\n\nThe Sorted Words Are:\n");
   for(i=0;i<n;++i)
      printf("%s\n",wrds[i]);
}

int check(char c)
{  switch(c)
   {
      case '.' :
      case ';' :
      case ',' :
      case '\t':
      case ' ' : return 1;
                 break;
      default  : return 0;
   }
}
```

70.

```
/* To check whether a string is palindrome or not */
#include<string.h>
#include<stdio.h>
int ispalindrome(char s1[]);
main()
{  char text[80];
   clrscr();
   printf("Enter Your Text : ");
   gets(text);
   if(ispalindrome(text))
      printf("%s is Palindrome " ,text);
   else
      printf("%s is not Palindrome ",text);
   getch();
}

int ispalindrome(char s1[])
{  int i,j;
   j=strlen(s1);
```

```
    for(i=0,j=j-1;i<=j;++i,--j)
        if(s1[i]!=s1[j])
            return 0;
    return 1;
}
```

UNIT VII

DATA FILES

7.1. Data Files

So far, we assumed that all data input to a program originate from the standard input device, namely the keyboard of a video terminal. We have also assumed that all output information is displayed on the standard output device, namely the screen of a video terminal. But many applications require that information be written to or read from an auxiliary memory device. Such information is stored on memory device in the form of a data file. Thus, the data files allow the users to store information permanently, and to access and alter that information whenever necessary.

The C language supports two different types of data files, called Stream-oriented (or Standard) data files, and System-oriented (or Low-level) data files. Stream-oriented files are generally easier to work with and are therefore more commonly used. In C, extensive set of library functions is available for creating and processing both types of these data files.

Stream-oriented data files can be subdivided into two categories - text files and unformatted (Binary) data files. The steam-oriented text files consist of consecutive characters. These characters can be interpreted as individual data items, or as components of string or numbers. The manner in which these characters are interpreted is determined either by particular library function used to transfer the information, or by format specification within the library functions, as in the printf and scanf functions.

The Stream-oriented unformatted data files organize data into blocks containing contiguous bytes of information. These blocks represent more complex data structures, such as arrays and structures. A separate set of library functions is available for processing stream-oriented unformatted data files. Theses library functions provide single instructions that can transfer entire arrays or structures to or from data files.

System-oriented data files are closely related to the computers operating system. They are somewhat more complicated to work with, though their use may be more efficient for certain kinds of applications. A separate set of library functions is available for system-oriented data files.

7.2. Opening and Closing a Stream-Oriented Data File

The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed. That is, the C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a stream, and the actual device is called a file.

The C file system is designed to work with a wide variety of devices including terminals, disk drivers, and tape drivers. Even though each device is very different, the buffered file system transforms each into a logical device called a stream. All streams behave similarly. Because streams are largely device independent, the same function that can write a disk file can also write to another type of device, such as a console.

In C, files may be anything from a disk file to a terminal or printer. One need to associate a stream with a specific file by performing an 'open' operation. Once file is open, information can be exchanged between the file and the program. The programmer disassociates a file from a specific stream with 'close' operation. If the program closes a file opened for output, the contents, if any, of its associated stream are written to the external device.

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 files more rapidly than would otherwise be possible. The buffer area is established by writing

```
FILE *ptvar;
```

where FILE (Upper-case 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. The structure type FILE is defined within the stdio.h file. The pointer ptvar is often referred to as stream pointer or simply a stream.

A data file must be opened before it can be created or processed. This associates the file name with the buffer area (i.e., with the steam). It also specifies how data file will be utilized, i.e., as read-only file, a write-only file, or a read/write file. The library function fopen is used to open a file. This function is typically written as

```
ptvar = fopen(file-name, file-type);
```

where the file-name and file-type are strings that represent the name of the data file and the manner in which the data file is utilized. The name chosen for the data file must be consistent with the rules for naming files.

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 found. Finally, a data file must be closed at the end of the program. A file can be closed by using the library function fclose. The syntax is simply

```
fclose(ptvar);
```

7.3. File Handling Functions

fopen

Declaration: FILE *fopen(const char *filename, const char *mode);

The fopen() function opens a file whose name is pointed to by 'filename' and returns the stream that is associated with it. The type of operations that will be allowed on the file are defined by the value of mode. The legal values of modes are shown in the following table.

File Type	Meaning
"r"	Open an existing file for reading only.
"w"	Open a new file for 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 (i.e., for adding new information at the end of the file). If a file with the specified file-name currently does not exist, a new file will be created.
"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. If a file with the specified file-name currently does not exist, a new file will be created.

If fopen() is successful in opening the specified file, a FILE pointer is returned. If the file cannot be opened, a NULL pointer is returned.
The following code uses fopen() to open a file named TEST for output.

```
FILE *fptr;
fptr = fopen("TEST", "w");
```

Although the preceding code is technically correct, the correct method of opening file is illustrated by the following code fragment.

```
FILE *fptr;
if (( fptr=open("TEST", "w") ) == NULL)
{  printf("Cannot Open File\n");
   exit(1);
}
```

This method detects any error in opening a file, such as write-protected or a full disk before attempting to write to it. If no file by that name exists, one will be created. Opening a file for read operations require that the file exists.

fclose

Declaration: int fclose(FILE *stream);
The fclose function closes the file associated with stream and flushes its buffer (i.e., it writes any data still remaining in the disk buffer to the file). After a call to fclose(), stream is no longer connected with the file, and any automatically allocated buffers are deallocated.

If fclose() is successful, zero is returned; otherwise EOF is returned. For example, the following code opens and close a file:

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   if ((fptr = fopen("TEST","w"))==NULL)
   {  printf("Cannot Open File.\n");
      exit(1);
   }
   if(fclose(fptr)) printf("File Close Error.\n");
}
```

fgetc

Declaration: int fgetc(FILE *stream);

The fgetc() function returns the next character from the specified input stream and increments the file position indicator. The character is read as an unsigned char that is converted to an integer. If the end-of-file is reached, fgetc() returns EOF. If fgetc() encounters an error, EOF is also returned. For example, the following program reads and displays the contents of a text file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char c;
   if ((fptr = fopen("TEST","r"))==NULL)
   {  printf("Cannot Open File.\n");
      exit(1);
   }
   while((c=fgetc(fptr)) != EOF)
      putchar(c);
   if(fclose(fptr)) printf("File Close Error.\n");
}
```

fputc

Declaration: int fputc(int ch, FILE *stream);

The fputc() function writes the character ch to the specified stream at the current file position and then advance the file position indicator. Even though the ch is declared to be an int, it is converted by fputc() into an unsigned char.

The value returned by the fputc() is the value of the character written. If an error occurs, EOF is returned.

For example, the following program writes a string to the specified stream.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char text[80];
   int i=0;
   clrscr();
   printf("Enter a Text : ");
   gets(text);
   if ((fptr = fopen("test","w"))==NULL)
   {  printf("Cannot Open File.\n");
      exit(1);
   }
   while(text[i]!='\0') fputc(text[i++],fptr);

   if(fclose(fptr)) printf("File Close Error.\n");
   getch();
}
```

getc

Declaration: int getc(FILE *stream)

The getc() function returns the next character from the specified input stream and increments the file position indicator. The character is read as an unsigned char that is converted to an integer.

If the end-of-file is reached, getc() returns EOF. If getc() encounters an error, EOF is also returned. The functions getc() and fgetc() are identical.

For example, the following program reads and displays the contents of a text file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char c;
   clrscr();
```

```
  if ((fptr = fopen("TEST","r"))==NULL)
  {  printf("Cannot Open File.\n");
     exit(1);
  }
  while((c=getc(fptr)) != EOF)
     putchar(c);
  if(fclose(fptr)) printf("File Close Error.\n");
  getch();
}
```

putc

Declaration: int putc(int ch, FILE *stream);

The putc() function writes the character ch to the specified stream at the current file position and then advance the file position indicator. Even though the ch is declared to be an int, it is converted by putc() into an unsigned char.

The value returned by the putc() is the value of the character written. If an error occurs, EOF is returned. The putc() and fputc() functions are identical.

For example, the following program writes a string to the specified stream.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char text[80];
   int i=0;
   clrscr();
   printf("Enter a Text : ");
   gets(text);
   if ((fptr = fopen("test","w"))==NULL)
   {  printf("Cannot Open File.\n");
      exit(1);
   }
   while(text[i]!='\0') putc(text[i++],fptr);
   if(fclose(fptr)) printf("File Close Error.\n");
   getch();
}
```

fgets

Declaration: char *fgets(char *str, int num, FILE *stream)

The fgets() function reads up to num-1 characters from stream and store them in the character array pointed to by str. Characters are read until either a newline or an EOF is received or until the specified limit is reached. After the character have been read, a null is stored in the array immediately after the last character is read. A newline character will be retained and will be a part of the array pointed to by str.

If successful, fgets() returns str; a null pointer is returned upon failure. If a read error occurs, the contents of the array pointed to by str are indeterminate.

For example, the following program uses fgets() to display the content of a file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char text[80];
   clrscr();
   if ((fptr = fopen("test","r"))==NULL)
   {  printf("Cannot Open File.\n");
      exit(1);
   }
   fgets(text,80,fptr);
   puts(text);
   if(fclose(fptr)) printf("File Close Error.\n");
   getch();
}
```

fputs

Declaration: int fputs(const char *str, FILE *stream)

The fputs function writes the contents of the string pointed to by str to the specified stream. The null terminator is not written. The fputs() function returns nonnegative on success and EOF on failure.

For example, the following program uses fputs() to write a string into a file.

```
#include<stdio.h>
#include<stdlib.h>
main()
{  FILE *fptr;
   char text[80];
   int i=0;
```

```
    clrscr();
    printf("Enter a Text : ");
    gets(text);
    if ((fptr = fopen("test","w"))==NULL)
    {  printf("Cannot Open File.\n");
       exit(1);
    }
    fputs(text,fptr);
    if(fclose(fptr)) printf("File Close Error.\n");
    getch();
}
```

fscanf

Declaration: fscanf(FILE *stream, const char *format, ...)

The fscanf function works exactly like the scanf function except that it reads the information from the stream specified by stream instead of standard input device.

fprintf

Declaration: int fprintf(FILE *stream, const char *format, ...)

The fprintf() function outputs the values of the arguments that makes up the argument list as specified in the format string to the stream pointed to by stream. The operations of the format control string and commands are identical to those in printf().

The following program uses the fprintf() and fscanf() functions to prepare the results of ‘n’ students.

```
/* To prepare the mark list*/
#include<stdio.h>
main()
{  int regno,m[5],tot,i,j,n,k;
   char name[15];
   float avg;
   FILE *fptr;
   if((fptr=fopen("mark","w")) == NULL)
   {  printf("Error in Opening File..");
      exit(1);
   }
   clrscr();
```

```
   printf("How many students ? ");
   scanf("%d", &n);
   for(i=0;i<n;++i)
   {  clrscr();
      printf("Enter the details of Student-%d\n\n",i+1);
      printf("Name ? ");
      scanf("   %[^\n]",name);
      printf("Reg. No ? ");
      scanf("  %d",&regno);
      for(tot=0,j=0;j<5;++j)
      {  printf("Mrak in Paper-%d ? ",j+1);
         scanf("%d",&m[j]);
         tot+=m[j];
      }
      avg=(float)tot/5.0;
      fprintf(fptr,"%15s %d %d %d %d %d %d %f\n",
                  name,regno,m[0],m[1],m[2],m[3],m[4],tot,avg);
   }
   clrscr();
   fclose(fptr);
   if((fptr=fopen("mark","r")) == NULL)
   {  printf("Error in Opening File..");
      exit(1);
   }
   printf("Name\t\tReg.No\tM1  M2  M3  m4  M5   Total  Avg   Res\n\n");
   for(i=0;i<n;++i)
   {  fscanf(fptr,"%-15s %d %d %d %d %d %d %d %f",
               name,&regno,&m[0],&m[1],&m[2],&m[3],&m[4],&tot,&avg);
      printf("%15s %4d %4d %4d %4d %4d %4d %4d %6.2f",
                  name,regno,m[0],m[1],m[2],m[3],m[4],tot,avg);
      if(avg>35.0)
         printf(" Passed");
      else
         printf(" Failed");
   }
   fclose(fptr);
   getch();
}
```

feof

Declaration: int feof(FILE *stream)

The feof() function determines whether the end-of-file associated with stream has been reached. A nonzero value is returned if the file position indicator is at the end-of-file; zero is returned otherwise.

rewind

Declaration: void rewind(FILE *stream)

The rewind() function moves the file position indicator to the start of the specified stream. It also clears the end-of-file and error flags associated with stream.

7.4. Command Line Arguments

In all our programs so far, main has been defined with an empty parameter list. However, it possible to pass arguments to main when the program is invoked for execution. These arguments are referred to as command line arguments. Thus, command line arguments are the information that follows the program's name on the command line of the operating system. For example, to display the contents a file in DOS environment, you type something like

type sample.c

where sample.c is a command line argument that specifies the name of the file you wish to display.

In C, two separate built-in arguments, argc (argument count) and argv (argument vector) are used to receive command line arguments. The argc parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The argv is an array of pointers to characters, i.e., an array of strings. Each string in this array will represent a parameter that is passed to main. All command line arguments are strings – any numbers will have to be converted by the program into the proper binary format, manually.

The function prototype for main can be thought of as

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

In order to pass one or more parameters to the program when it is executed from the operating system, the parameters must follow the program name on the command line, e.g.,

pgm-name parameter1 parameter2 parametern

The individual items must be separated from one another either by blank spaces or by tabs.

The program name will be stored in as the first item in the argv, followed by each parameters. Hence, if the program name is followed by n parameters, there will be (n+1) entries in argv, ranging from argv[0] to argv[n]. Moreover, argc will automatically be assigned the value (n+1). Note that the value for argc is not supplied explicitly from the command line.

For example, consider the following program, which will be executed from a command line.

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

This program allows an unspecified number of parameters to be entered from the command line. When the program is executed, the current value of argc and the elements of argv will be displayed as separate lines of output. Suppose, for example, that the program name is sample, and the command line initiating the program execution is

sample one two three four

Then the program will be executed, resulting the following output.

```
argc = 5
argv[0]=sample
argv[1]=one
argv[2]=two
argv[3]=three
argv[4]=four
```

Once the parameters are entered, they can be utilized within the program in any desired manner. One particularly common application is to specify the names of the data files as command line parameters, as illustrated below.

Suppose that we wanted to write a program called create which, when invoked, creates a file. Thus, for example, the command line

create sample.doc <contents terminated by EOF character ^Z>

will creates a file named sample.doc with the contents that follows the file name and ends with the end-of-file character Ctrl+Z.

```
#include<stdio.h>
main(int argc, char *argv[])
{  FILE *fptr;
   char c;
   clrscr();
   if ((fptr = fopen(argv[1],"w"))==NULL)
```

```
  {  printf("Cannot Open File.\n");
     exit(1);
  }
  while((c=getchar()) != EOF)
     putc(c,fptr);
  if(fclose(fptr)) printf("File Close Error.\n");
}
```

Questions

3 Mark Question

1. What is the primary advantage of using a data file?
2. Describe the different ways in which the data files can be categorized in C.
3. What is a stream? How does a stream differ from a file?
4. What is the purpose of a buffer area when working with a stream oriented data file? How is a buffer defined?
5. What are different modes of opening a stream oriented file? Explain each.
6. Explain the purpose of fopen and fclose functions.
7. Explain the purpose of fgetc and fputc functions.
8. Explain the purpose of getc and putc functions.
9. Explain the purpose of fgetc and fputc functions.
10. Explain the purpose of fgets and fputs functions.
11. Explain the purpose of fscanf and fprintf functions.
12. What is the use of feof function? Illustrate with an example.
13. What is the use of rewind function? Illustrate with an example.
14. What are command line arguments? Explain its purpose.
15. What are command line arguments? Explain the purposes of argc and argv parameters.

8 Mark Question

1. What is a data file? Explain the procedure to use a data file in C. Describe various file oriented library functions in C with suitable example.

Programming Questions

1. To create an employee database file containing Employee Name, Code, Basic, D.A and P.F. Display the salary details of a given employee. Give a suitable error message if the given employee's details does not exist in the database.
2. To input Name, Reg.No and marks in five subjects for a set of students into a data file. Display the result of a particular student. Give a suitable error message if the given student's details does not exist in the data file.

3. To input the inventory details such as Item name, Item code, Quantity, Price, into a data file. Display the details of a given item. Give a suitable error message if the given item's details does not exist in the data file.
4. To create a text file-the file name and the contents of the file being the command line arguments.
5. To create a 'cpy' command that is equivalent to the copy command of the DOS-name of the source file and the destination file being the command line arguments.
6. To create a 'display' command that is equivalent to the type command of the DOS-name of the file being a command line argument.
7. To search a string in a file-the file name and the string to be searched being the command line arguments.
8. To create a 'charcount' command that will count the number of characters in a file-the name of the file being a command line argument.
9. To merge the contents of two files into a third file-the names of the files being command line arguments.
10. To compare the contents of two files on a character-by-character basis. The results of comparisons should be placed into a file that contains a sequence of 0s and 1s. A 1 is placed into the file if the corresponding characters of the two files are equal. A 0 is placed otherwise-file names being the command line arguments.
11. To input a set of numbers and place the odd and even numbers into separate files.
12. To find the value of Sin(θ), where θ be a command line argument.
13. To find the value of Cos(θ), where θ be a command line argument.
14. To create an 'add' command that can add two numbers given as command line arguments.
15. To create an 'add' command that can add any set of numbers that are given as command line arguments (e.g., add 10 20 30 40 etc).

Answers to Selected Questions

(Flowcharting is left as an exercise to the students)

1.

```
/* To create the inventory file */
#include<stdio.h>
#include<string.h>
main()
{  int i,itemno,qty,flag,n;
   char name[15],nam[15],c;
   float price,tot;
   FILE *fptr;
   if((fptr=fopen("item","w")) == NULL)
   {  printf("Error in Opening File..");
      exit(1);
   }
   clrscr();
   printf("How many items ? ");
   scanf("%d", &n);
   for(i=0;i<n;++i)
   {  clrscr();
      printf("Enter the details of Item-%d\n\n",i+1);
      printf("Item Name ? ");
      scanf("   %[^\n]",name);
      printf("Item No ? ");
      scanf("  %d",&itemno);
      printf("Price ? ");
      scanf("%f",&price);
      printf("Quantity ? ");
      scanf("%d",&qty);
      fprintf(fptr,"%15s %d %d %f\n",name,itemno,qty,price);
   }
   if(fclose(fptr)) printf("File Close Error.\n");
   clrscr();
  if((fptr=fopen("item","r")) == NULL)
   {  printf("Error in Opening File..");
      exit(1);
   }
   do
   {  rewind(fptr);
      clrscr();
      flag = 0;
```

```
      printf("Name of item to be displayed ? ");
      scanf("  %[^\n]",nam);
      for(i=0;i<n;++i)
      {  fscanf(fptr,"%15s %d %d %f\n",name,&itemno,&qty,&price);
         puts(name);
         if(strcmpi(name,nam) == 0)
         {  printf("Name\t\tItem No\tQuantity\tPrice\tTotal\n");
            tot = qty*price;
            printf("%-15s\t%d\t%d\t\t%.2f\t%.2f\n",name,itemno,qty,price,tot);
            flag = 1;
            break;
         }
      }
      if(!flag)
         printf("The item %s does not exist in the file",name);
      printf("\n\n\nDisplay More (Y/N) ? ");
      c=getche();
   }
   while(toupper(c) == 'Y');
   if(fclose(fptr)) printf("File Close Error.\n");
}
```

5.

/* To create a command ‘cpy’ that is equivalent to the copy command of DOS. Assume that this program is stored in a file named ‘cpy’. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

cpy filename1 filename2

This will copy the contents of the filename1 to filename2. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1,*fptr2;
   char c;
   if(argc>3)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<3)
      {  printf("Too few parameters ...");
         exit(1);
      }
```

```
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("Error in Opening the Source File %s ...",strupr(argv[1]));
      exit(1);
   }
   if ((fptr2 = fopen(argv[2],"w"))==NULL)
   {  printf("Unable To Create the Destination File %s ...",strupr(argv[2]));
      exit(1);
   }
   while(!feof(fptr1))
      putc(c=getc(fptr1),fptr2);
   if(fclose(fptr1)) printf("File Close Error.\n");
   if(fclose(fptr2)) printf("File Close Error.\n");
   printf("%s Successfully copied to %s\n",strupr(argv[1]),strupr(argv[2]));
}
```

6.

/* To create a command 'display' that is equivalent to the type command of DOS. Assume that this program is stored in a file named 'display'. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

display filename

This will display the contents of the filename on standard output device */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1;
   char c;
   if(argc>2)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<2)
      {  printf("Too few parameters ...");
         exit(1);
      }
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[1]));
      exit(1);
   }
   while(!feof(fptr1))
      putchar(c=getc(fptr1));
```

```
   if(fclose(fptr1)) printf("File Close Error.\n");
}
```

7.

/* To create a command 'searchstring', that searches file to check whether a string exists in a file or not. Assume that this program is stored in a file named 'searchstring'. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

searchstring filename search-string

The search-string should be enclosed in double quotation marks, if it contains more than one word. This will display the count of the string */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1;
   char c;
   int flag = 1,count=0,i=0,len;
   if(argc>3)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<3)
      {  printf("Too few parameters ...");
         exit(1);
      }
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[1]));
      exit(1);
   }
   len= strlen(argv[2]);
   while(!feof(fptr1))
   {
      c=getc(fptr1);
      if (c != argv[2][i++])
         {flag = 0;i=0;}
      else
         flag = 1;
      if(i==len)
         if(flag)
            {  count++;
```

```
            i=0;
          }
    }
    if(count>0)
       printf("The Word \"%s\" exists %d times in File \"%s\" ",
argv[2],count,argv[1]);
    else
       printf("The Word \"%s\" does not exists in File \"%s\"  ",
argv[2],argv[1]);
    if(fclose(fptr1)) printf("File Close Error.\n");
}
```

8.

/* To create a command 'charcount', that counts the number of characters in a file. Store the program in a file named charcount. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

charcount filename

This will display the count of the number of characters in the file */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1;
   char c;
   float count = 0;
   if(argc>2)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<2)
      {  printf("Too few parameters ...");
         exit(1);
      }
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[1]));
      exit(1);
   }
   while(!feof(fptr1))
   {  c=getc(fptr1);
      count++;
   }
   printf("The File \"%s\" contains %g characters ", argv[1],count);
```

```
   if(fclose(fptr1)) printf("File Close Error.\n");
}
```

9.

/* Merge the contents of two files. Store the program in a file named merge. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

merge filename1 filename2 filename3

This will merge the contents of filename1 and filename2 to filename3. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1,*fptr2,*fptr3;
   char c;
   if(argc>4)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<4)
      {  printf("Too few parameters ...");
         exit(1);
      }
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("\a\a\aError in Opening File %s ...",strupr(argv[1]));
      exit(1);
   }
   if ((fptr2 = fopen(argv[2],"r"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[2]));
      exit(1);
   }
   if ((fptr3 = fopen(argv[3],"w"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[3]));
      exit(1);
   }
   while(!feof(fptr1))
   {  c=getc(fptr1);
      putc(c,fptr3);
   }
  if(fclose(fptr1)) printf("File Close Error.\n");
   while(!feof(fptr2))
```

```
   {  c=getc(fptr2);
      putc(c,fptr3);
   }
   if(fclose(fptr2)) printf("File Close Error.\n");
   if(fclose(fptr3)) printf("File Close Error.\n");
   printf("Files \"%s\" and \"%s\" are successfully merged to File \"%s\"",
      argv[1],argv[2],argv[3]);
}
```

10.

/* Check the contents of two files and creates a third file that contains a sequence of 1s and 0s. Store the program in a file named check. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

check filename1 filename2 filename3

This will check the contents of filename1 and filename2 on a character-by-character basis and add 1 or 0 to filename3. */

```
#include<stdio.h>
#include<string.h>
main(int argc, char *argv[])
{  FILE *fptr1,*fptr2,*fptr3;
   char c1,c2;
   if(argc>4)
      {  printf("Too many parameters ...");
         exit(1);
      }
   else
      if(argc<4)
      {  printf("Too few parameters ...");
         exit(1);
      }
   if ((fptr1 = fopen(argv[1],"r"))==NULL)
   {  printf("\a\a\aError in Opening File %s ...",strupr(argv[1]));
      exit(1);
   }
   if ((fptr2 = fopen(argv[2],"r"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[2]));
      exit(1);
   }
   if ((fptr3 = fopen(argv[3],"w"))==NULL)
   {  printf("Error in Opening File %s ...",strupr(argv[3]));
      exit(1);
```

```
   }
   while ( (!feof(fptr1)) || (!feof(fptr2)))
   {
      if(!feof(fptr1))
         c1=getc(fptr1);
      else
         c1='\0';
      if(!feof(fptr2))
        c2=getc(fptr2);
      else
         c1='\0';
      if (c1==c2)
         putc('1',fptr3);
      else
         putc('0',fptr3);
   }
   if(fclose(fptr1)) printf("File Close Error.\n");
   if(fclose(fptr2)) printf("File Close Error.\n");
   if(fclose(fptr3)) printf("File Close Error.\n");
   printf("Files \"%s\" and \"%s\" are successfully Checked and created File \"%s\"",
         argv[1],argv[2],argv[3]);
}
```

13.

/* To Sum the Cosine series. Store the program in a file named Cos. Compile the program and execute it from the command mode of the operating system. The command line initiating the program would look like

cos angle

This will display the value of Cos(θ) */

```
#include<stdio.h>
#include<math.h>
double fact(int n);
double power(float x, int n);
main(int argc, char *argv)
{
   int i;
   float sign;
   double x,tempx,sum=1.0,term=1.0;
   if(argc>2)
   {  printf("Too many arguments ...");
      exit(1);
   }
```

```
   if(argc<2)
   {  printf("Too few arguments ...");
      exit(1);
   }
   sscanf(argv[1],"%f",&x);
   printf("x=%f",x);
   tempx=x;
   x=x*3.14/180.0;
   sign = -1;
   for(i=2;fabs(term)>0.0001;i+=2)
   {  term=sign*power(x,i)/fact(i);
      sum+=term;
      sign = -sign;
   }
   printf("Cos(%g) = %.2f",tempx,sum);
   }
double fact(int n)
{  int i;
   double f = 1.0;
   for(i=2;i<=n;++i)
   f*=i;
   return f;
}
double power(float x, int n)
{  double p=1,i;
   for(i=1;i<=n;++i)
   p=p*x;
   return p;
}
```

References

1. Computer System Architecture - M Morris Mano
2. Computer Organization and Architecture – William Stallings
3. Computer Fundamentals – P.K Sinha
4. Programming With C – Byron Gottfried
5. Programming With C – Venugopal and Prasad
6. Let Us C – Yashvant Kanetkar
7. Programming in C – Kernighhan & Ritchie
8. The Spirit of C – Mullish Cooper

MODEL QUESTION PAPERS

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **ten** of the following questions*

1. Convert:
   (a) $(127)_{10}$ to hexadecimal
   (b) $(326)_8$ to binary
2. Explain the signed binary representation with reference to 52 and -23.
3. Explain the binary subtraction
4. Explain parity code.
5. Explain XNOR.
6. Explain the truth table for JK Flip-Flop.
7. Give the concept of RISK
8. Explain DMA
9. Explain the use of scanf in C.
10. Explain the storage class in C
11. Explain pointers in C
12. Define Structure and union.

(10×2 = 20 Marks)

PART B

*Answer any **three** of the following questions*

13. Explain the working of primary memory (RAM, ROM and EPROM) and their specific uses in computers.
14. (a). Explain the CPU and system bus.
    (b) Explain *two* addressing modes.
15. (a). Draw the realization of $F_1=aB+Ab$ using the minimum number of NAND gates.
    (b). Simplify using K-map $F_2=\sum m(0,2,5,7,8,10,11,12,15)$
16. (a). Write a C program to sum N numbers.
    (b). Explain array. Explain how it is entered in a computer.
17. (a). Explain the function prototyping. What are its advantages?
    (b).Write a C program to sum the series (1)+(1+2)+(1+2+3)+… up to *n* terms.
18. (a). Explain structure and discuss its advantages.
    (b). Write a C program to calculate simple interest and list it as a look-up table.

(3×10=30 Marks)

Note: This was the question paper pattern used for 2007 March examination, which is different from the one mentioned in the syllabus.

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **ten** of the following questions*

1. State the advantages of using computers:
2. What is a parity bit?
3. How a NAND gate is implemented using AND gates only?
4. Distinguish between RAM and ROM.
5. How a register is formed?
6. What are I/O devices?
7. What is an algorithm?
8. Distinguish between *int* and *log int* operators in C language.
9. Give the use and syntax of scanf statement in C language.
10. What is the use of unary operators in C programs?
11. Distinguish between local and global variables.
12. What is a pointer?

(10×2½ = 25 Marks)

PART B

*Answer any **two and half** of the following questions*

13. (a). Explain the features of any two computer generations.
    (b). How errors are detected and corrected in data recording?
14. (a). How a K-Map is constructed from a truth table?.
    (b). Distinguish between CD and DVD..
15. With the help of a block diagram .explain the functions different components of a processor.
16. (a). Draw a flowchart to find the factorial of a given number.
    (b). Write a C program to find the factorial of a given number.
17. (a). Explain the syntax and use of any two control statements in C language?
    (b). Write a C program to store values in a 3×3 array and to print the sum of values in each row and column.

(2 ½×10 = 25 Marks)

Note: This was the question paper pattern used for 2006 May examination, which is different from the one mentioned in the syllabus.

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Convert the following numbers into decimal numbers:
   (a). $(110110)_2$ (b). $(573)_8$
2. Differentiate between a bit, a byte and a word.
3. What is a cache memory? How is it different from a primary memory?
4. Describe the basic logic gates.
5. Define an array and list its importance.
6. Discuss the features of a good program.
7. Discuss the standard library functions of C Language and their uses.
8. Differentiate local and global variables.

(6×3=18 Marks)

PART B

*Answer any **four** of the following questions*

1. Explain the components of central processing unit and describe their functions.
2. Explain the following terms: -
   (a) RAM (b) ROM (c) Magnetic disk (d) Virtual memory.
3. State and prove any two theorems of Boolean Algebra.
4. Describe DMA technique.
5. Explain with examples any three looping statements of C language.
6. (a). Discuss the input and output functions of C language.
   (b). Define a variable. Explain its different types.

(4×8=32 Marks)

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Convert $(8B2F)_{16}$ to decimal and octal.
2. Explain universal gates..
3. What is RAM? Explain its different types.
4. Differentiate between RISC and CISC.
5. What is an interrupt? Explain.
6. Describe the if…else structure in C. Explain its use?
7. What is pointers? What are its advantages?.
8. What is a structure? How does it differ from an array?.

(6×3=18 Marks)

PART B

*Answer any **four** of the following questions*

1. Write a program in C language to accept a set of numbers, which contains +ve and –ve numbers. Display the sum and average of +ve and negative numbers separately.
2. Write a program to sort a set of numbers.
3. Explain rules for binary arithmetic operation in digital computers.
4. Realize the simplified circuit for $f=\Sigma(0,2,4,5,7,8,10,13,14,15)$.
5. Explain two types of secondary storage devices.
6. Describe the hardwired and microprogrammed control units..

(4×8=32 Marks)

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

(2005 Admission)

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Convert the decimal number 247.68 into its octal and binary equivalents.
2. Draw the combinational circuit diagram of a full adder. Also write its truth table.
3. Explain the concept of micro programmed control unit.
4. Simplify the Boolean function F=*x'yz+x'yz'+xy'z'+xy'z* using K-Map
5. Differentiate between impact and non-impact printers.
6. What are the different classes of statements in C?
7. Write the syntaxes of printf and scanf functions. Also explain their use.
8. What is a register variable?

(6×3=18 Marks)

PART B

*Answer any **four** of the following questions*

1. Discuss briefly about the various secondary storage devices.
2. How does magnetic tape differ from magnetic disk ? Explain
3. Explain in the following terms: -
   (a) Automatic Variables (b) External Variables
4. Write a program to read two matrices from the user and to add those matrices .
5. List and explain various functions used in files.
6. Discuss briefly about the evolution of computers.

(4×8=32 Marks)

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

(2005 Admission)

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Perform the following binary subtractions using 1's and 2's complement representations.
   (a). 1010010-1101111 (b). 111100-111011
2. What is meant by parity bit? Explain how parity is used for error detection.
3. State and prove De Morgan's theorem.
4. Why is NAND gate called Universal gate? Justify.
5. Differentiate between hardwired and micro-programmed control units.
6. What is meant by instruction pipelining? Illustrate with an example.
7. Differentiate between argument passing by reference and by value.
8. Differentiate between a structure and a Union in C.

(6×3=18 Marks)

Part B

*Answer any **three** of the following questions*

1. Describe the magnetic disk system.
2. What is an instruction set? Describe the classification of instructions in a typical computer.
3. What is virtual memory? Explain demand paging.
4. Describe the three different I/O techniques employed in digital computers.
5. Describe the various types of printers.
6. Draw a flowchart to create a mark data file, which contains the mark details of 'n' students appeared for an examination. The file contains fields for Reg.No, Name, and Marks in five subjects. Now search the data file to display the result of a particular student. Also write a C program for the flowchart you just prepared.

(4×8=32 Marks)

First Year B.Sc. Degree Examination

Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

(2005 Admission)

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Perform the following operations of signed binary numbers using 1's and 2's complement representations.
   (a). 1,1101+1,110011
   (b). 1,10111-0,101
2. What is mean by biased exponent? Explain its advantage.
3. List and explain the memory hierarchy in a typical computer system.
4. Differentiate between RISC and CISC.
5. What is Program Counter? Explain its function.
6. Explain the differences between
   (a). 'a' and "a" (b). a++ and ++a (c). a and &a
7. What is a static variable? How does it differ from external and automatic variables?
8. What is structure in C? How does it differ from an array? How are structures defined?

(6×3=18 Marks)

Part B

*Answer any **four** of the following questions*

1. Briefly describe the basic functional components of a computer with the help of a block diagram.
2. Discuss any three number systems along with rules for converting from one number system to another. Illustrate with an example.
3. What is virtual memory? Describe demand paging.
4. What is a full adder? Design a full-adder circuit with logic gates.
5. What is parallel processing? Describe different forms of parallel processing.
6. (a). Draw a flowchart to display the first 'n' prime numbers.
   (b). Write a C program to find the transpose of a matrix.

(3×10=30 Marks)

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary
Paper I – Computer Organization and Programming in C

(2005 Admission)

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** of the following questions*

1. Convert the numbers $(1011.11)_2$, $(1024.3)_8$ and $(5c2.01)_{16}$ into decimal number system.
2. Perform the following binary arithmetic.
   a. 1100.011+1011.011 b. 1100011.001-110111.11 c. 1100110000÷111
3. What is Excess-3 code? Give its code sequence.
4. Define the Boolean operators AND, OR and NOT. Obtain their truth tables.
5. What is Cache memory? Explain its functions.
6. List the functional units of a CPU. Explain their functions.
7. What RISC? List and explain its characteristics.
8. Explain two different forms of referring array elements.

(6×3=18 Marks)

Part B

*Answer any **four** of the following questions*

1. Discuss the rules for signed binary addition/subtraction of binary numbers. Illustrate with examples.
2. a. State and explain the basic laws of Boolean algebra.
   b. Describe various logic gates used in digital computers.
3. What is mean by addressing mode? List and explain various addressing modes used in digital computers.
4. What are peripheral devices? How are they classified? Briefly describe one device from each class.
5. What is recursion? Explain with an example
6. Draw a flowchart to input two matrices and to find their product. Display the matrices entered along with the product matrix. Also write a C program for the flowchart you prepared.

(10×3=30 Marks)

First Year B.Sc. Degree Examination
Part III – Computer Application – Subsidiary

Paper I – Computer Organization and Programming in C

Time: Three Hours Maximum: 50 Marks

Part A

*Answer any **six** Questions*

1. Perform the following base conversions using shortcut method.
   (a) $(FACE.BAD)_{16}=(?)_2$ (b) $(10110111.1)_2=(?)_{16}$ (c) $(567.321)_8 = (?)_8$
2. List and explain different ways of representing negative numbers.
3. What is a Truth table? Give example.
4. What is meant by addressing mode? List various addressing modes used in digital computers.
5. What is an Algorithm? Explain the necessary characteristics of an algorithm.
6. What are relational and logical operators? List the relational and logical operators used in C.
7. What is meant by operator precedence and associativity? Summarise the precedence and associativity of C operators.
8. What are pointers? How are pointers declared and used?

(6×3=18 Marks)

Part B

*Answer any **four** Questions*

1. Discuss various computer generations describing their special characteristics.
2. Simplify the Boolean function F(A,B,C,D)=ab+Ac+c+ad+aBc+abc using K-Map. Realize the simplified expression using:
   (1) NAND gates only and (2) NOR gates only
3. What is ROM? Describe different types of ROM.
4. List the secondary storage devices used in computers. Briefly explain their organization and characteristics.
5. Draw a flowchart to find and display the complete roots of quadratic equation. Also write a C program for the same.
6. Draw a flowchart to sort a set of strings. Also write C program for the same.

(4×8=32 Marks)