(2½ Hours) **[Total Marks: 75]**

N. B.: (1) **All** questions are **compulsory**.
(2) Make**suitable assumptions** wherever necessary and **state the assumptions** made.
(3) Answers to the **same question** must be **written together**.
(4) Numbers to the **right** indicate **marks**.
(5) Draw **neat labeled diagrams** wherever **necessary**.
(6) Use of **Non-programmable** calculators is **allowed**.

1. Attempt *any three* of the following:

a. Explain the different types of programming language.

There are many different languages can be used to program a computer. The most basic of these is ***machine language--a*** collection of very detailed, cryptic instructions that control the computer's internal circuitry. Thisis the natural dialect of the computer. Very few computer programs are actually written in machine language,however, for two significant reasons: First, because machine language is very cumbersome to work with andsecond, because every different type of computer has its own unique instruction set. Thus, a machine-languageprogram written for one type of computer cannot be run on another type of computer without significantalterations.

Usually, a computer program will be written in some ***high-level*** language, whose instruction set is more compatible with human languages and human thought processes. Most of these are ***general-purpose*** languages such as C. (Some other popular general-purpose languages are Pascal, Fortran and BASIC.) There are also various ***special-purpose*** languages that are specifically designed for some particular type of application. Some common examples are CSMP and SIMAN, which are special-purpose ***simulation*** languages, and LISP, a ***Zistprocessing***
language that is widely used for artificial intelligence applications.

As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. This greatly simplifies the task of writing complete, correct programs. Furthermore, the rules for programming in a particular high-level language are much the same for all computers, so that a program written for one computer can generally be run on many different computers with little or no alteration. Thus, we see that a high-level language offers three significant advantages over machine language: ***simplicity, uniformity*** and ***portability*** (i.e., machine independence).

A program that is written in a high-level language must, however, be translated into machine language before it can be executed. This is known as ***compilation*** or ***interpretation,*** depending on how it is carried out. (Compilers translate the entire program into machine language before executing any of the instructions.Interpreters, on the other hand, proceed through a program by translating and then executing singleinstructions or small groups of instructions.) In either case, the translation is carried out automatically within the computer. In fact, inexperienced programmers may not even be aware that this process is taking place,since they typically see only their original high-level program, the input data, and the calculated results. Most implementations of C operate as compilers.

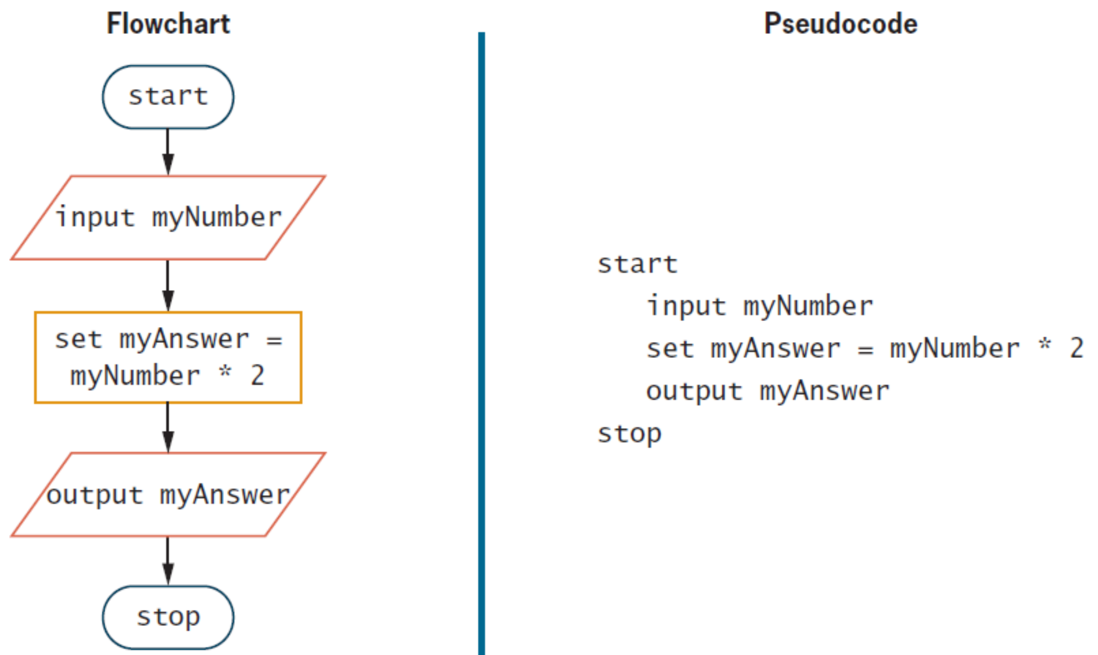
A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, and generates a corresponding machine-language program as output. The original high-level program is called the ***source*** program, and the resulting machine-language program is called the ***object*** program. Every computer must have its own compiler or interpreter for a particular high-level language.It is generally more convenient to develop a new program using an interpreter rather than a compiler. Once an error-free program has been developed, however, a compiled version will normally execute much faster than an interpreted version.

b. Explain the different steps in the program development cycle.

The steps of the **program development cycle** are as follows:

1. Understand the problem.
2. Plan the logic.
3. Code the program.
4. Use software (a compiler or interpreter) to translate the program into machine language.
5. Test the program.
6. Put the program into production.
7. Maintain the program.

c. Draw the flowchart and pseudo code of program that doubles a number.



d. Describe the structure of a C program.

Every C program consists of one or more modules calledfunctions. One of the functions must be called main. The program will always begin by executing the main function, which may access other functions. Any other function definitions must be defined separately, either ahead of 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.
The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as parameters.)
Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called expression statements) and other compound statements. Thus compound
statements may be nested, one within another. Each expression statement must end with a semicolon (;).Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters / * and */ (e.g., /* t h i s is a comment */). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

```
/* program to calculate the area of a c i r c l e */ /* TITLE (COMMENT) */
#include <stdio.h> / * LIBRARY FILE ACCESS */
main( ) / * FUNCTION HEADING */
{
f l o a t radius, area; /* VARIABLE DECLARATIONS */
p r i n t f ("Radius = ? / * OUTPUT STATEMENT (PROMPT) * I ' I ) ;
scanf ( "%'fI , &radius) ; I * INPUT STATEMENT * /
area = 3.14159 * radius * radius; /* ASSIGNMENT STATEMENT */
p r i n t f ("Area = %f",area) ; / * OUTPUT STATEMENT */
}
```

e. What are the various data types in c? Explain them.
C supports several different types of data, each of which may be represented differently within the computer’s

memory. The basic data types are listed below. Typical memory requirements are also given. (The memory
requirements for each data type will determine the permissible range of values for that data type. Note that the
memory requirements for each data type may vary from one C compiler to another.)

Built in

i n t integer quantity 2 bytes or one word (varies fromone compiler to another)
char single character 1 byte
f l o a t floating-point number (i.e., a number containing 1 word **(4** bytes) a decimal point andor an exponent)
double double-precision floating-point number (i.e., more 2 words (8 bytes)significant figures, and an exponent which maybe larger in magnitude)

Derived types

They include
(a) Pointer types,
(b) Array types,
(c) Structure types,
(d) Union types and
(e) Function types.
The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return

f. What is a statement in C? Explain the different classes of statement in C.

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 and
control statements.
An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.
EXAMPLE Several expression statements are shown below.

```
a = 3;
c = a + b ;
++i;
p r i n t f ("Area = %f area) ;
```

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. Thus, the compound statement provides **a** capability for embedding statements within other statements. Unlike an expression statement, a compound statement does ***not*** end with a semicolon.
EXAMPLE A typical compound statement is shown below.

```
{
p i= 3.141593;
circumference= 2. * p i* radius;
area = p i* radius * radius;
}
```

Control statements are used to create special program features, such as logical tests, loops and branches. Many control statements require that other statements be embedded within them, as illustrated in the following
example.
EXAMPLE The following control statement creates a conditional loop in which several actions are executed repeatedly, until some particular condition is satisfied.

```
while (count <= n) {
p r i n t f ( ' x = ' ) ;
scanf( "Skf" , &x) ;
sum += x;
++count;
}
```

2. Attempt *any three* of the following: 15

a. Write a program in C to swap two numbers without using third variable.

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

```
{
int a, b;
printf ("Enter First Number: ");
scanf("%d", &a);
printf ("Enter Second Number: ");
scanf("%d", &b);
b=b-a;
a=b+a;
b=a-b;
printf ("\nNew Value of First Number: %d", a);
printf ("\nNew Value of Second Number: %d", b);
}
```

b. Describe the five arithmetic operators in C.

There are five ***arithmetic operators*** in C. They are

Operator	**Purpose**
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after integer division

The % operator is sometimes referred to as the ***modulus operator.***

There is no exponentiation operator in C. However, there is a ***library function*** **(POW)** to carry out exponentiation**.**The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands canbe integer quantities, floating-point quantities or characters (remember that character constants representinteger values, as determined by the computer's character set). The remainder operator (%) requires that both 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 a 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.

EXAMPLE 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**

c. Explain the conditional operator in C.

Simple conditional operations can be carried out with the ***conditional operator*** **(?** :). ***An*** expression that makes use of the conditional operator is called a ***conditional expression.*** Such an expression **can** be written in place of the more traditional **if -else** statement.

A conditional expression is written in the form

expression1 ? ***expression 2*** : ***expression 3***

When evaluating a conditional expression, ***expression 1***is evaluated first. If ***expression 1*** is true (i.e., if its value is nonzero), then ***expression*** 2 is evaluated and this becomes the value of the conditional expression. However, if ***expression 1*** is false (i.e., if its value is zero), then ***expression*** 3is evaluated and this becomes the value of the conditional expression. Note that only one of the embedded expressions (either ***expression 2*** or ***expression 3)*** is evaluated when determining the value of a conditional expression.

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

d. Explain the getchar and putchar functions used in C programming language.

THE getcharFUNCTION

Single characters can be entered into the computer using the C library function **getchar.**

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

In general terms, a function reference would be written **as**

character variable = **getchar**( ) ;

where***character variable*** refers to some previously declared character variable.

EXAMPLE

A C program contains the following statements.

char c;

. . . . .
c = getchar();

THE putcharFUNCTION

Single characters can be displayed (ie, written out of the computer) using the C library function **putchar.** This function is complementary to the character input function **getchar**

Example

The **putchar**function, like **getchar,** is a part of the standard C **I/O** library. It transmits a single character to a standard output device .The character being transmitted will normally be represented **as** a character-type variable. It must be expressed as an argument to the function, enclosed in
parentheses, following the word **putchar.**

putchar(***character variable)***

where***character variable*** refers to some previously declared character variable.

EXAMPLE

A C program contains the following statements.

char c;

. . . . .
putchar(c);

e. Write a short note on scanf function.

Input data can be entered into the computer from a standard input device by means of the C library function ***scanf.*** This function can be used to enter any combination of numerical values, single characters and strings. The function returns the number **of** data items that have been entered successfully.In general terms, the ***scanf***function is written **as *scanf(contro1 string, argl, arg2,*** . . . , ***argn)***

where***control string*** refers to a string containing certain required formatting information, and ***argl,arg2,*** . . . ***argn***are arguments that represent the individual input data items.

The control string consists of individual groups 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 willconsist of the percent sign, followed by a
conversion character which indicates the type of the corresponding data item.Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace characters (i.e., blank spaces, tabs or newline characters). If whitespace characters are used to separate multiple character groups in the control string, then all consecutive whitespace characters in the input data will be read but ignored. The use of blank spaces as character-group separators is very common

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, hexadecimal or octal integer
0 data item is an octal integer
S data item is a string followed by a whitespace character (the null character \ ***0*** will automatically be added at the end)
U data item is **an** unsigned decimal integer
X data item is a hexadecimal integer
[...] data item is a string which may include whitespace characters

The arguments are written **as** variables or arrays, whose types match the corresponding character groupsin the control string. ***Each variable name must be preceded by an ampersand*** (&).

Example

char item(201;
i n t partno;
f l o a t cost;
. . . . .
scanf("%s %d %f",item, Bpartno, &cost);

…………………

f. Explain the gets and puts functions used in C programming language.

The gets() function reads string from user and puts() function prints the string. Both functions are defined in <stdio.h> header file.

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

Each of these functions accepts a single argument. The argument must be a data item that represents a string. (e.g., a character array). The string may include whitespace characters. In the case of gets, the string will be entered from the keyboard, and will terminate with a newline character (i.e., the string will end when the user presses the Enter key).

```
#include<stdio.h>
#include <string.h>
int main(){
char name[50];
   printf("Enter your name: ");
   gets(name); //reads string from user
   printf("Your name is: ");
   puts(name);  //displays string
 return 0;
}
```

3. Attempt *any three* of the following: **15**

a. Explain if-else statement with an example.

The if-else statement in C language is used to execute the code if condition is true or false.

The if - else 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 else portion of the if - else statement is optional. Thus, in its simplest general form, the statementcan be written as
if (expression) statement
The expression must be placed in parentheses, as shown. In this form, the statement will be executedonly if the expression has a nonzero value (i.e., if expression is true). If the expression has a value ofzero (i.e., if expression is false), then the statement will be ignored.
The statement can be either simple or compound. In practice, it is often a compound statement whichmay include other control statements.

The syntax of if-else statement is given below:

```
if(expression){
        //code to be executed if condition is true
        }else{
        //code to be executed if condition is false
```

Example

```
#include<stdio.h>
int main(){
int number=0;
```

```
printf("enter a number:");
scanf("%d",&number);
if(number%2==0){
printf("%d is even number",number);
}
else{
printf("%d is odd number",number);
}
return 0;
}
```

b. Write a program in C to find the sum of squares of digits of a number.

```
#include <stdio.h>
main()
{
int a, tot=0, b;
printf ("\nEnter a number: ");
scanf ("%d", &a);
while (a>0)
{
b=(a%10);
tot=tot+(b*b);
a=a/10;
}
printf("\nSum of squares of digits is %d\n", tot);
}
```

c. What is the difference between while and do-while loop in C.

While loop	Do While loop
while is an ENTRY CONTROLLED LOOP	do while is an EXIT CONTROLLED LOO
while (expression) statement The statement will be executed repeatedly, as long as the expression is true (that is as long expression has a nonzero value). This statement can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the expression, thus providing a stopping condition for the loop	do statement while (expression); The statement will be executed repeatedly, as long as the value of expression is true (i.e., is nonzero). Notice that statement will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The statement can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of expression so the looping action can terminate.
The statements of the loop execute based on the condition Condition to be satisfied is a true condition	The statements of the loop execute atleast once Condition to be satisfied is a false Condition
Eg	Eg

```
#include<stdio.h>  #include<stdio.h>
int main(){    int main(){
int i=1;      int i=1;
while(i<=10){                               do{
printf("%d \n",i);      printf("%d \n",i);
i++;   i++;
}                                           }while(i<=10);
return 0;                                   return 0;
}                                            }
```

d. Explain function with an example.

A function is a self-contained program segment that carries out some specific, well-defined task. Every C program consists of one or more functions **.** One of these functions must be called **main.**
Execution of the program will always begin by carrying out the instructions in **main.** Additional functions will be subordinate to **main,** and perhaps to one another.
If a program contains multiple fuctions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another.
A function will carry out its intended action whenever it is ***accessed*** (i.e., whenever the function is "called") from some other portion of the program. 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.
Generally, ***a finction will process information that is passed to it fiom the calling portion of the program, and return a single value.*** Information is passed to the function via special identifiers called ***arguments*** (alsocalled ***parameters),*** and returned via the **r eturn**statement. Some functions, however, accept infonnation butdo not return anything (as, for example, the library function **p r i n t f) ,** whereas other functions (e.g., thelibrary function **scanf)** return multiple values.

```
#include <stdio.h>
char lower-to-upper(char c l ) /* function d e f i n i t i o n */
char c2;
c2 = ( c l >= ' a ' &&c l <= '2 ')7 ( ' A ' + c l - ' a ' ) : c l ;
return(c2);
}
main( )
{
char lower, upper;
printf("P1ease enter a lowercase character: " ) ;
scanf( *%c" &lower) ;
upper = lower-to-upper( lower) ;
p r i n t f ( \nThe uppercase equivalent is  %c\n\n", upper) ;
}
```

e. Write a program in C to find the factorial of a number using recursion.

```
#include <stdio.h>
intn_fact(int n);
main()
{
int n, res;
printf ("Enter number to find factorial: ");
scanf ("%d", &n);
res=n_fact(n);
printf ("Factorial is %d", res);
}
intn_fact (int n)
{
int result;
if ( n == 0 )
```

```
result = 1;
else
result = n * n_fact ( n-1 );
return (result);
}
```

f. Explain call by value and call by reference.

Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include <stdio.h>
void call_by_value(int x) {
printf("Inside call_by_value x = %d before adding 10.\n", x);
x += 10;
printf("Inside call_by_value x = %d after adding 10.\n", x);
}
int main() {
int a=10;
printf("a = %d before function call_by_value.\n", a);
call_by_value(a);
printf("a = %d after function call_by_value.\n", a);
return 0;
}
```

The output of this call by value code example will look like this:

a = 10 before function call_by_value.
Inside call_by_value x = 10 before adding 10.

Call by Reference

In this method the reference variables of the actual arguments are created as formal arguments. Reference variables are aliases of the original variables and refer to the same memory locations as the original variables do. Since the formal arguments are aliases of the actual arguments, if we do any changes in the formal arguments they will affect the actual arguments.

```
#include <stdio.h>
void call_by_value(int &x) {
printf("Inside call_by_value x = %d before adding 10.\n", x);
x += 10;
printf("Inside call_by_value x = %d after adding 10.\n", x);
}
int main() {
int a=10;
printf("a = %d before function call_by_value.\n", a);
call_by_value(a);
printf("a = %d after function call_by_value.\n", a);
return 0;
}
```

4. Attempt *any three* of the following: 15

a What is meant by the storage class of a variable?

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 different storage-class specifications in C: ***automatic, external, static*** and ***register.*** They are identified by the keywords auto, extern, s t a t i c , and r e g i s t e r , 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 variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

External variables, in contrast to automatic variables, are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. Hence, they usually span two or more functions, and often an entire program. They are often referred to as ***global variables.*** Since 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.

Static variables are defined within a function in the same manner as automatic variables, except that the variable declaration must begin with the **s t a t i c** storage-class designation. Static variables can be utilized within the function in the same manner as other variables. They cannot, however, be accessed outside of their defining function

Registers are special storage areas within the computer's central processing unit. The actual arithmetic and logical operations that comprise a program are carried out within these registers. 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 values of ***register variables*** are stored within the registers of the central processing unit. A variable can be assigned this storage class simply by preceding the type declaration with the keyword ***register*** .

b Write short note on Global variable.

External variables scope extends from the point of definition through the remainder of the program. Hence, they usually span two or more functions, and often an entire program. They are often referred to as ***global variables.*** Since 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 especially convenient when a function requires numerous input data items. Moreover, we now have a way to transfer multiple data items out of a function, since the r e t u r n statement can return only one data item.

When working with external variables, we must distinguish between external variable ***definitions*** and external variable ***declarations.*** An external variable ***definition*** **is** written in the same manner **as** an ordinary variable declaration. It must appear outside of, and usually before, the functions that access the external variables. An external variable definition will automatically allocate the required storage space for the external variables within the computer's memory. The storage-class specifierextern 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 function.

On the other hand, if the function definition ***precedes*** the external variable definition, then the function must include a ***declaration*** for that external variable.

c Write a program in C to calculate successive Fibonacci numbers.

```
#include <stdio.h>
long i n t fibonacci(int count);
main( )
{
i n t count, n;
printf("How many Fibonacci numbers? ' ) ;
scanf( '%d", an) ;
p r i n t f ( " \ n " ) ;
```

```
f o r (count = 1; count <= n; ++count)
p r i n t f ( " \ n i = %2d F = %ld', count, fibonacci(count));
}
long i n t fibonacci(int count)
/* calculate a Fibonacci number using the formulas
F = 1 f o r i<3, and F = F1 + F2 f o r i>= 3 */
{
s t a t i c long i n t f l = 1, f 2 = 1;
long i n t f;
f = (count <3)  ?1 : f l + f2;
f 2 = fl;
f l = f;
r e t u r n ( f ) ;
}
```

d What are preprocessor directives in C? Explain #include and #define in C.

The C preprocessor is a collection of special statements, called directives, that are executed at the beginning of the compilation process. The **#include** and **#define** statements are preprocessor directives. Additional preprocessor directives are **#if, #elif, #else, #endif, #if def,#if ndef, #line** and **#undef.** The preprocessor also includes three special operators: **defined,** #, and ##.Preprocessor directives usually appear at the beginning of a program, though this is not a **firm** requirement. Thus, a preprocessor directive may appear anywhere within a program. However, the directive will apply only to the portion of the program following its appearance

The preprocessor statement **#include;** i.e.,

#include <*filename*>

where***filename*** represents the name of a special file. The names of these special files are specified by each individual implementation of C, though there are certain commonly used file names such as **stdio. h, s t d l i b . h**and **math. h.** The suffix **"h"** generally designates a "header" file, which indicates that it is to be included at the beginning of the program.

A symbolic constant is defined by writing

#define *name text*

where***name*** represents a symbolic name, typically written in uppercase letters, and ***text*** represents the sequence of characters that is associated with the symbolic name. Note that ***text*** does not end with a semicolon, since a symbolic constant definition is not a true C statement. Moreover, if ***text*** were to end with a semicolon, this semicolon would be treated **as** though it were a part of the numeric constant, character constant or string constant that is substituted for the symbolic name.

#define TAXRATE 0.23
#define PI 3.141593
#define TRUE 1
#define FALSE 0
#define FRIEND "Rahul"

e Write a program in C to arrange the ‘n’ numbers stored in the array in ascending order.

```
#include <stdio.h>
main(void)
{
int a[10], i=0, j=0, n, t;
printf ("\n Enter the no. of elements: ");
scanf ("%d", &n);
printf ("\n");
for (i = 0; i<n; i++)
{
printf ("\n Enter element %d: ", (i+1));
scanf ("%d", &a[i]);
}
for (j=0; j<(n-1); j++)
{
for (i=0; i<(n-1); i++)
{
if (a[i+1] < a[i])
{
t = a[i];
a[i] = a[i + 1];
a[i + 1] = t;
}
}
}
printf ("\n Ascending order: ");
for (i=0; i<n; i++)
{
printf (" %d", a[i]);
}
}
```

f What is a two dimensional array? How can they be declared and initialized in C.

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
storage-class data- type* array[*expression 1] [***expression* 2]**
an **m** x **n,** two-dimensional array can be thought of as a ***table*** of values having **m** rows and **n** columns,
float table[50][50];
char page[24][80];
The first line defines table as a floating-point array having 50 rows and 50 columns (hence 50 x 50 = 2500 elements),and the second line establishes page as a character array with 24 rows and 80 columns (24 x 80 = 1920 elements).
int values[3][4] = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
Note that **values** can be thought of **as** a table having **3** rows and **4** columns **(4** elements per row). Since the initial values are assigned by rows (i.e., last subscript increasing most rapidly), the results of this initial assignment are **as** follows.
values[O][O] = 1 values[O][l] = 2 values[0][2] = 3 values[0][3] = 4
values[l][O] = 5 values[l][l] = 6 values[l][2] = 7 values[l][3] = *8*
values[2][0] = 9 values[2][1] = 10 values[2][2] = 11 values[2][3] = 12
Remember that the first subscript ranges from 0 to **2,** and the second subscript ranges from 0 to **3.**

```
int values[3][4] = {
{1, 2, 3, 4},
{ 5, 6, 7, 8},
{ 9, 10, 11, 12}
};
```

5. Attempt *any three* of the following: **15**

a. Explain the term pointers with an example.

A ***pointer*** is a variable that represents the ***location*** (rather than the ***value)*** of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point.
In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.
Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements. Moreover, pointers provide a convenient way to represent multidimensional arrays, allowing a single multidimensional array to be replaced by a lower-dimensional array of pointers. This feature permits a group of strings to be represented within a single array, though the individual strings may differ in length.
Suppose v is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can then be accessed if we know the location (i.e., the ***address)*** of the first memory cell.* The address of v 's memory location can be determined by the expression &v, where &is a unary operator, called the ***address operator,*** that evaluates the address of its operand. Now let 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 memory. Remember, however, that pvrepresents v's ***address,*** not its value. Thus, pvis referred to as a ***pointer variable.***
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 (i.e., the contents of the same memory cells). Furthermore, if we write **pv= &v** and u = ***pv,** then u and **v** will both represent the same value; Le., the value of **v** will indirectly be assigned to u.

```
#include <stdio.h>
main( ){
i n t u = 3;
i n t v;
i n t *pu; /* pointer t o an integer */
```

```
i n t *pv; /* pointer t o an integer */
pu= &U; / * assign address of U t o pu*/
v = *pu; / * assign value of U t o v */
pv = &v; /* assign address of v t o pv*/
printf("\nu=%d &u=%X pu=%X *pu=%d', U, &U, pu, *pu);
printf("\n\nv=%d &v=%X pv=%x *pv=%d", V, &v, pv, *pv);
}
```

b. Write a C program to perform addition of two pointer variable.

```
#include <stdio.h>
main()
{
int first, second, *p, *q, sum;
printf ("Enter two integers to add\n");
scanf ("%d%d", &first, &second);
p = &first;
q = &second;
sum = *p + *q;
printf ("Sum of entered numbers = %d\n", sum);
}
```

c. Write a short note on pointer arithmetic.

A pointer in c is an address, which is a numeric value. Therefore, you can perform

arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -
To understand pointer arithmetic, let us consider that **ptr**is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer –
ptr++
After the above operation, the **ptr**will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr**points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.
Similarly ptr --
ptr decrements 4 bytes in case of interges and one byte in case of characters
Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

d. Differentiate between structure and union.
5 points 5 marks

e. What is an array within the structure and array of structure?
An array which is a part of the a structure is an element declared within a structure
In the example below it is char name[80]
struct date {
int month;
int day;
int year;
};
struct account {
int acct-no;

```
char acct-type;
char name[80];
f l o a t balance;
s t r u c t 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 (i.e., each element of customer represents an individual customer record).
Note that each structure of type account includes an array (name[80]) and another structure (date) as members.Thus, we have an array and a structure embedded within another structure, which is itself an element of an array.

f. Explain nested structure in C with an example.

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded or nested structure must appear before the declaration of the outer structure.

```
struct date {
int month;
int day;
int year;
};
struct account {
int acct-no;
char acct-type;
char name[80];
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 account.

```
oldcustomer = (12345, ' R I , "John W. Smith", 586.30, 5, 24, 90);
```

The first member (acct-no) is assigned the integer value 12345, the second member (acct-type) is assigned the character ' R I ,the third member (name [801) is assigned the string John W. Smith ",and the fourth member (balance) is assigned the floating-point value 586.30. 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, 24 and 90.