CS8251 - PROGRAMMING IN C

UNIT-I

PART A

1. What are the various types of operators?

   C supports the following types of operators

   Unary

Unary Plus	+
Unary Minus	-
Increment	++
Decrement	--

   Binary

Arithmetic	+,-,*,/,%
Relational	<,>,<=,>=,==,!=
Logical	&&,||,!
Bitwise	&,|,^,<<,>>
Assignment	=
Shorthand assignment	+=, -=,*=,/=,%=

   Ternary

   ?:

   Special

   Sizeof(), *. ->

2. Write a for loop to print from 10 to 1

```
for(i=10;i>0;i--)
    printf("%d",i);
```

3. Write any two preprocessor directives in C

   Preprocessor directive instructions are used to instruct the preprocessor to expand/translate the source program before compilation

   The two types of preprocessor directives are,

#define	to define Macro , Global constant
#include	to include library files

4. List different datatypes available in C

   Data types are used to specify the type of a variable. In c, the data types are classified into 3 category. They are,

Primary or Built-in	:	int , char, float
Derived	:	array, enum, pointer
User defined	:	function, structure

5. Write a C program to find factorial of a given number using Iteration

```
void main()
    {
    int N=5,I,fact=1;
    for(i=1;i<=N;i++)
        fact=fact*i;
    printf("The factorial value of %d is =%d",N,fact);
    }
```

6. What is the use of preprocessor directive?

- Used to translate one high level language statement into another high level language statement.
- Used to instruct the preprocessor that how it performs the translation before compilation.
- Used to perform conditional compilation.
- Used to include other C files or library files.

7. What is the variable? Illustrate with an example
   - Variable is a named storage area
   - Used to assign a name to a value
   - To declare a variable, we need to specify the data type of the value and the variable name

     Data type variable _name ;
   - Example

     int reg; float avg;

8. Define static storage class
   - Storage class specifies the visibility & life time of a variable
   - Static storage class initializes a variable to 0.
   - Longer life time – Exists throughout the execution of program
   - Visibility to all-Any function can access the variable- Shared by all

9. What is the use of #define preprocessor

   The #define preprocessor directive is used to define a

   Global constant #define PI 3.14

   Macro #define big(a,b) a>b?a:b

   When a compiler reads the macro name, it replaces the macro name by its expansion. When it reads the constant label, it replaces the label by its value.

10. What is the importance of keywords in C?

    Keywords are reserved identifiers

    They performs a specific task , specifies the data type

    Keyword can not be used as an identifier

    Ex: for , if , int

11. List out various Input & output statements in C

    The input & output statements are classified into formatted & unformatted I?O

    Formatted I/O : User can able to design/format the output

    Unformatted I/O: doesn't allow the users to design the output

Type	**Input**	**Output**
Formatted	scanf()	printf()
Unformatted	Getch(), getche() getchar() gets()	putch(), putchar() puts()

12. What is meant by linking process?

    Linking is a process of binding the function call to its definition and binding the label to its reference. During the linking process the object file is created. This file can be loaded into memory for execution

13. What is external storage class?

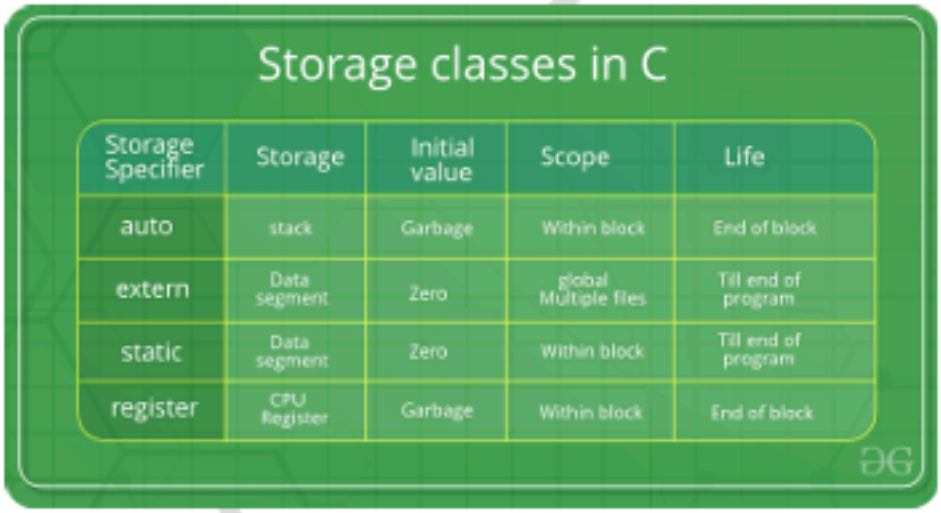
Ans: Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

14.What is the difference between structure and union.

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

PART B

1.Explain the storage classes in c with example programs.



A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

- The variable scope.
- The location where the variable will be stored.
- The initialized value of a variable.
- A lifetime of a variable.

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

```
Example, auto int age;
```

The program below defines a function with has two local variables

```
int add(void) {
  int a=13;
  auto int b=48;
return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

```
#include <stdio.h>
int main( )
{
 auto int j = 1;
 {
  auto int j= 2;
  {
   auto int j = 3;
   printf ( " %d ", j);
  }
  printf ( "\t %d ",j);
 }
 printf( "%d\n", j);}
```

OUTPUT:

```
3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

```
Example, extern void display();
```

First File: main.c

```
#include <stdio.h>
extern i;
main() {
  printf("value of the external integer is = %d\n", i);
  return 0;}
```

Second File: original.c

```
#include <stdio.h>
i=48;
```

Result:

```
value of the external integer is = 48
```

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

- Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.
- Static global variables are global variables visible **only to the file in which it is declared.**

```
Example: static int count = 10;
```

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

```
#include <stdio.h> /* function declaration */
void next(void);
static int counter = 7; /* global variable */
main() {
 while(counter<10) {
     next();
     counter++;   }
return 0;}
void next( void ) {    /* function definition */
  static int iteration = 13; /* local static variable */
  iteration ++;
  printf("iteration=%d and counter= %d\n", iteration, counter);}
```

Result:

```
iteration=14 and counter= 7
iteration=15 and counter= 8
iteration=16 and counter= 9
```

Global variables are accessible throughout the file whereas static variables are accessible only to the particular part of a code.

The lifespan of a static variable is in the entire program code. A variable which is declared or initialized using static keyword always contains zero as a default value.

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

```
Example: register int age;
```

The keyword **register** is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <stdio.h> /* function declaration */
main() {
{register int  weight;
int *ptr=&weight ;/*it produces an error when the compilation occurs ,we cannot get a memory location when dealing with CPU register*/}
}
```

OUTPUT:

```
error: address of register variable 'weight' requested
```

The next table summarizes the principal features of each storage class which are commonly used in C programming

Storage Class	Declaration	Storage	Default Initial Value	Scope	Lifetime
Auto	Inside a function/block	Memory	Unpredictable	Within the function/block	Within th function/block
register	Inside a function/block	CPU Registers	Garbage	Within the function/block	Within th function/block
Extern	Outside all functions	Memory	Zero	Entire the file and other files where the variable is declared as extern	program runtime
Static (local)	Inside a function/block	Memory	Zero	Within the function/block	program runtime

Static (global)	Outside functions	all	Memory	Zero	Global	program runtime

2. Explain the decision making statement in c with example programs.

DECISION STATEMENTS

It checks the given condition and then executes its sub-block. The decision statement decides the statement to be executed after the success or failure of a given condition.

Types:

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if....else statement
3. Nested if....else statement
4. Using else if stateme

Simple if statement

The general form of a simple if statement is,

```
if(expression)
{
   statement inside;
}
   statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>


void main( )
{
   int x, y;
   x = 15;
   y = 13;
   if (x > y )
   {
      printf("x is greater than y");
   }
}
```

```
x is greater than y
```

if...else statement

The general form of a simple if...else statement is,

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

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

```c
#include <stdio.h>


void main( )
{
   int x, y;
   x = 15;
   y = 18;
   if (x > y )
   {
      printf("x is greater than y");
   }
   else
   {
      printf("y is greater than x");
   }
}
```

```
y is greater than x
```

Nested if....else statement

The general form of a nested if...else statement is,

```c
if( expression )
{
   if( expression1 )
   {
      statement block1;
   }
   else
   {
```

```
        statement block2;
    }
}
else
{
    statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first if to perform the check for the next if block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

Example:

```
#include <stdio.h>


void main( )
{
    int a, b, c;
    printf("Enter 3 numbers...");
    scanf("%d%d%d",&a, &b, &c);
    if(a > b)
    {
        if(a > c)
        {
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
```

```
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
}
```

else if ladder

The general form of else-if ladder is,

```
if(expression1)
{
    statement block1;
}
else if(expression2)
{
    statement block2;
}
else if(expression3 )
{
    statement block3;
}
else
```

```
    default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

```
#include <stdio.h>
void main( )
{
   int a;
   printf("Enter a number...");
   scanf("%d", &a);
   if(a%5 == 0 && a%8 == 0)
   {
      printf("Divisible by both 5 and 8");
   }
   else if(a%8 == 0)
   {
      printf("Divisible by 8");
   }
   else if(a%5 == 0)
   {
      printf("Divisible by 5");
   }
   else
   {
      printf("Divisible by none");
   }
}
```

3. Explain the looping statement in c with example programs.

LOOP CONTROL STATEMENTS

Loop is a block of statements which are repeatedly executed for certain number of times. Types
1. For loop
2. Nested for loops
3. While loop
4. do while loop
5. do-while statement with while loop

'C' programming language provides us with three types of loop constructs:

1. The while loop

2. The do-while loop

3. The for loop

While Loop

A while loop is the most straightforward looping structure. The basic format of while loop is as follows:

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

It is an entry-controlled loop. In while loop, a condition is evaluated before processing a body of the loop. If a condition is true then and only then the body of a loop is executed. After the body of a loop is executed then control again goes back at the beginning, and the condition is checked if it is true, the same process is executed until the condition becomes false. Once the condition becomes false, the control goes out of the loop.

After exiting the loop, the control goes to the statements which are immediately after the loop. The body of a loop can contain more than one statement. If it contains only one statement, then the curly braces are not compulsory. It is a good practice though to use the curly braces even we have a single statement in the body.

In while loop, if the condition is not true, then the body of a loop will not be executed, not even once. It is different in do while loop which we will see shortly.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num=1;          //initializing the variable
        while(num<=10)   //while loop with condition
        {
                printf("%d\n",num);
                num++;              //incrementing operation
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program illustrates the use of while loop. In the above program, we have printed series of numbers from 1 to 10 using a while loop.

1. We have initialized a variable called num with value 1. We are going to print from 1 to 10 hence the variable is initialized with value 1. If you want to print from 0, then assign the value 0 during initialization.
2. In a while loop, we have provided a condition (num<=10), which means the loop will execute the body until the value of num becomes 10. After that, the loop will be terminated, and control will fall outside the loop.
3. In the body of a loop, we have a print function to print our number and an increment operation to increment the value per execution of a loop. An initial value of num is 1, after the execution, it will become 2, and during the next execution, it will become 3. This process will continue until the value becomes 10 and then it will print the series on console and terminate the loop.

\n is used for formatting purposes which means the value will be printed on a new line.

Do-While loop

A do-while loop is similar to the while loop except that the condition is always executed after the body of a loop. It is also called an exit-controlled loop.

The basic format of while loop is as follows:

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

As we saw in a while loop, the body is executed if and only if the condition is true. In some cases, we have to execute a body of the loop at least once even if the condition is false. This type of operation can be achieved by using a do-while loop.

In the do-while loop, the body of a loop is always executed at least once. After the body is executed, then it checks the condition. If the condition is true, then it will again execute the body of a loop otherwise control is transferred out of the loop.

Similar to the while loop, once the control goes out of the loop the statements which are immediately after the loop is executed.

The critical difference between the while and do-while loop is that in while loop the while is written at the beginning. In do-while loop, the while condition is written at the end and terminates with a semi-colon (;)

The following program illustrates the working of a do-while loop:

We are going to print a table of number 2 using do while loop.

```
#include<stdio.h>
#include<conio.h>
int main()
{
	int num=1;		//initializing the variable
	do		//do-while loop
	{
		printf("%d\n",2*num);
		num++;			//incrementing operation
	}while(num<=10);
	return 0;
}
```

Output:

```
2
4
6
8
10
12
14
16
18
20
```

In the above example, we have printed multiplication table of 2 using a do-while loop. Let's see how the program was able to print the series.

1. First, we have initialized a variable 'num' with value 1. Then we have written a do-while loop.

2. In a loop, we have a print function that will print the series by multiplying the value of num with 2.

3. After each increment, the value of num will increase by 1, and it will be printed on the screen.

4. Initially, the value of num is 1. In a body of a loop, the print function will be executed in this way: 2*num where num=1, then 2*1=2 hence the value two will be printed. This will go on until the value of num becomes 10. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

For loop

A for loop is a more efficient loop structure in 'C' programming. The general structure of for loop is as follows:

```
for (initial value; condition; incrementation or decrementation )
{
  statements;
}
```

- The initial value of the for loop is performed only once.
- The condition is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- The incrementation/decrementation increases (or decreases) the counter by a set value.

Following program illustrates the use of a simple for loop:

```
#include<stdio.h>
int main()
{
        int number;
        for(number=1;number<=10;number++)        //for loop to print 1-10 numbers
        {
                printf("%d\n",number);           //to print the number
        }
        return 0;
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The above program prints the number series from 1-10 using for loop.

1. We have declared a variable of an int data type to store values.
2. In for loop, in the initialization part, we have assigned value 1 to the variable number. In the condition part, we have specified our condition and then the increment part.
3. In the body of a loop, we have a print function to print the numbers on a new line in the console. We have the value one stored in number, after the first iteration the value will be incremented, and it will become 2. Now the variable number has the value 2. The condition will be rechecked and since the condition is true loop will be executed, and it will print two on the screen. This loop will keep on executing until the value of the variable becomes 10. After that, the loop will be terminated, and a series of 1-10 will be printed on the screen.

In C, the for loop can have multiple expressions separated by commas in each part.

For example:

```
for (x = 0, y = num; x < y; i++, y--) {
  statements;
}
```

Also, we can skip the initial value expression, condition and/or increment by adding a semicolon.

For example:

```
int i=0;
int max = 10;
for (; i < max; i++) {
  printf("%d\n", i);
}
```

Notice that loops can also be nested where there is an outer loop and an inner loop. For each iteration of the outer loop, the inner loop repeats its entire cycle.

Consider the following example, that uses nested for loops output a multiplication table:

```
#include <stdio.h>
int main() {
int i, j;
int table = 2;
int max = 5;
for (i = 1; i <= table; i++) { // outer loop
  for (j = 0; j <= max; j++) { // inner loop
    printf("%d x %d = %d\n", i, j, i*j);
  }
  printf("\n"); /* blank line between tables */
}}
```

Output:

```
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5

2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
```

The nesting of for loops can be done up-to any level. The nested loops should be adequately indented to make code readable. In some versions of 'C,' the nesting is limited up to 15 loops, but some provide more.

4.Explain operators inC

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

ARITHMETIC OPERATORS

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	**Description**
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division
++	Increment operator - increases integer value by one

--	Decrement operator - decreases integer value by one

Relational operators

The following table shows all relation operators supported by C.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

C language supports following 3 logical operators. Suppose a = 1 and b = 0,

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double(These are datatypes, we will learn about them in the next tutorial).

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now lets see truth table for bitwise &, | and ^

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted

Assignment Operators

Assignment operators supported by C language are as follows.

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b

+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is :

```
expression 1 ? expression 2: expression 3
```

Explanation:

- The question mark "**?**" in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of " **:** " i.e (expression 2) is executed.

- If (expression 1) returns false then the expression on the right side of " : " i.e (expression 3) is executed.

Special operator

Operator	Description	Example
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x** ; return address of the variable **x**
*	Pointer to a variable	***x** ; will be pointer to a variable **x**

5.Explain preprocessor directive.

Introduction

- A program which processes the source code before it passes through the compiler is known as **preprocessor.**
- The commands of the preprocessor are known as **preprocessor directives.**
- It is placed before the main().
- It begins with a # symbol.
- They are never terminated with a semicolon.

Preprocessor Directives

The preprocessor directives are divided into four different categories which are as follows:

1. Macro expansion

- There are two types of macros - one which takes the argument and another which does not take any argument.
- Values are passed so that we can use the same macro for a wide range of values.

Syntax:

```
#define name replacement text
```

Where,
name – it is known as the micro template.
replacement text – it is known as the macro expansion.

- A macro name is generally written in capital letters.
- If suitable and relevant names are given macros increase the readability.
- If a macro is used in any program and we need to make some changes throughout the program we can just change the macro and the changes will be reflected everywhere in the program.

Example : Simple macro

```
#define LOWER 30
void main()
{
    int i;
    for (i=1;i<=LOWER; i++)
    {
        printf("\n%d", i);
    }
}
```

Example : Macros with arguments

```
#define AREA(a) (3.14 * a * a)
void main()
{
    float r = 3.5, x;
    x = AREA (r);
    printf ("\n Area of circle = %f", x);
}
```

Some of the predefined macros which are readily available are as follows:

Macro	Description
___LINE___	It contains a current line number as a decimal constant.
___FILE___	It contains the current filename as a string literal.
___DATE___	It shows the current date as a character literal in the “MMM DD YYYY” format.
___TIME___	It shows the current time as a character literal in “HH:MM:SS” format.

___STDC___	It is defined as 1 when the compiler complies with the ANSI standard.
___TIMESTAMP___	It is a sing literal in the form of "DDD MM YYYY Date HH:MM:SS". It is used to spec and time of the last modification of the current source file.

2. File inclusion

- The file inclusion uses the #include.

Syntax:

#include filename

- The content that is included in the filename will be replaced at the point where the directive is written.
- By using the file inclusive directive, we can include the header files in the programs.
- Macros, function declarations, declaration of the external variables can all be combined in the header file instead of repeating them in each of the program.
- The stdio.h header file contains the function declarations and all the information regarding the input and output.

There are two ways of the file inclusion statement:

i) #include "file-name"

ii) #include <file-name>

- If the first way is used, the file and then the filename in the current working directory and the specified list of directories would be searched.
- If the second way, is used the file and then the filename in the specified list of directories would be searched.

3. Conditional compilation

- The conditional compilation is used when we want certain lines of code to be compiled or not.
- It uses directives like #if, #elif, #else, #endif

Syntax

```
#if TEST <= 5
statement 1;
statement 2;
#else
statement 3;
statement 4;
#endif
```

If there are a number of conditions to be checked we can use the #elif instead of #else and #if.

4. Miscellaneous directive

There are some directives which do not fall in any of the above mentioned categories.

There are two directives:

i) #undef : This directive is used in relation to the #define directive. It is used to undefine a defined macro.
ii) #pragma : It is a specialized and rarely used directive. They are used for turning on and off certain features.

Summary of preprocessor directives

Following table will show you various directives that we have studied in this chapter:

Directives	Description
#define	It substitutes a preprocessor macro.
#include	It inserts a particular header file from another file.
#undef	A preprocessor macro is undefined.
#ifdef	It returns true if the macro is defined.
#ifndef	It returns true if the macro is not defined.
#if	It tests if the compile time condition is true.
#else	It is an alternative for #if.
#elif	It has #else and #if in one statement.
#endif	The conditional preprocessor is ended.
#error	It prints the error message on stderr.
#pragma	It issues special commands to the compiler by using a standardized method.

UNIT- II
PART A

1. Define Array
   Array is a collection of similar type of values
   All values are stored in continuous memory locations
   All values share a common name
   Linear data structure. The elements are organized in a sequential order.

2. Name any two library functions for handling string
   strlen() – finds the length of a string. It returns an integer value. It counts the no. of characters except null character & returns the count
   strlen(str)
   strcpy() – copies the source string into destination string. So, the source string should be enough to store the destination string.
   strcpy(source,destination)

3. Declare a float array of size 5 and assign 5 values to it
   Declaration : float price[5];
   Initialization : float price[5]={200.50,150.25,25.5,55.75,40.00}; (or)
   float price[]={1.2,3.4,6.5,7.8,9.8};

4. Give an example for initialization of string array
   String is a character array.
   Collection of one or more characters- enclosed with in double quotes
   Declaration : char name[10];
   Initialization : char name[10]="India";
   car name[10]={'I','n','d','i','a'};
   The char array is terminated by '\0'

5. How a character array is is declared
   Declaration : char name[n];
   This array can store n-1 characters.
   Initialization : char name[10]="India";
   car name[10]={'I','n','d','i','a'};
   The char array is terminated by '\0'

6. Write example code to declare two dimensional array
   Two dimensional array is an array with two subscript values. First subscript specifies the row & second subscript specifies the column. Used to process matrix operations.
   Declaration : datatype array_name [r][c];
   int matrixA[10][10];
   This matrixA can store 100 elements in a row major order.

7. What is mean & median of a list of elements?
   Mean : Average of the N elements can be computed by
   sum of N elements/N
   Ex: 2,1,3,4,5
   Mean = (2+1+3+4+5)/5 = 3
   Median : Middle element of a list. To find the median, the list must be sorted first.
   If N is odd then Median= (N+1)/2
   Ex: 1,2,3,4,5
   Median= 6/2 . The element at 3rd position is the median

   If N is even then then Median is the average of
   Median= (a[(N+1)/2]+a[(N-1)/2])/2
   Ex: 1,2,3,4,5,6
   Median=(a[3]+a[2])/2
   =4+3/2
   =3.5

8. Define Searching
   Searching is a process of finding the position of a given element in a list. The searching is successful if the element is found. There are two types of searching.
   Linear Search
   Binary Search

9. Define Sorting
   Sorting is a process of arranging the elements either in ascending order or descending order.

10. Sort the following elements using selection sort method. 23,55,16,78,2

Step1:Find smallest element in the list & exchange the element with first element of the list

2,55,16,78,23

Step2: Find second smallest value & exchange it with the second element of the list

2,16,55,78,23

Step 3: Continue the process until all the elements are arranged in the order

2,16,23,78,55

Step 4: 2,16,23,55,78

PART B

1.Write a c program to multiply two matrices (2D array) which will be entered by a user.

Ans:

```
#include <stdio.h>

int main()
{
    int m, n, p, q, c, d, k, sum = 0;
    int first[10][10], second[10][10], multiply[10][10];

    printf("Enter number of rows and columns of first matrix\n");
    scanf("%d%d", &m, &n);
    printf("Enter elements of first matrix\n");

    for (c = 0; c < m; c++)
        for (d = 0; d < n; d++)
            scanf("%d", &first[c][d]);

    printf("Enter number of rows and columns of second matrix\n");
    scanf("%d%d", &p, &q);

    if (n != p)
        printf("The multiplication isn't possible.\n");
    else
    {
        printf("Enter elements of second matrix\n");

        for (c = 0; c < p; c++)
            for (d = 0; d < q; d++)
                scanf("%d", &second[c][d]);

        for (c = 0; c < m; c++) {
            for (d = 0; d < q; d++) {
                for (k = 0; k < p; k++) {
                    sum = sum + first[c][k]*second[k][d];
                }

                multiply[c][d] = sum;
                sum = 0;
            }
        }
```

```
                printf("Product of the matrices:\n");

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++)
                    printf("%d\t", multiply[c][d]);

                printf("\n");
                }
                }

                return 0;
    }
```

2.Write a c program to find scaling of two matrices (2D array) which will be entered by a user.(5)
Ans:

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int disp[2][3];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
printf("enter the value to be scaled");
scanf("%d",&z);
      printf("Enter value for disp[%d][%d]:", i, j);
      scanf("%d", &z*disp[i][j]);
    }
     }
```

3.Write a c program to find determinant of two matrices (2D array) which will be entered by a user.

```
#include <stdio.h>

void main()
  {
 int arr1[10][10],i,j,n;
 int det=0;


    printf("\n\nCalculate the determinant of a 3 x 3 matrix :\n");
    printf("---------------------------------------------\n");

       printf("Input elements in the first matrix :\n");
    for(i=0;i<3;i++)
     {
       for(j=0;j<3;j++)
       {
              printf("element - [%d],[%d] : ",i,j);
              scanf("%d",&arr1[i][j]);
```

```
        }
      }
         printf("The matrix is :\n");
         for(i=0;i<3;i++)
         {
          for(j=0;j<3 ;j++)
            printf("% 4d",arr1[i][j]);
           printf("\n");
         }

 for(i=0;i<3;i++)
    det = det + (arr1[0][i]*(arr1[1][(i+1)%3]*arr1[2][(i+2)%3] -
arr1[1][(i+2)%3]*arr1[2][(i+1)%3]));

 printf("\nThe Determinant of the matrix is: %d\n\n",det);
    }
```

4.explain Matrix addition program.

```
  #include <stdio.h>
int main() {
int r, c, a[100][100], b[100][100], sum[100][100], i, j;
printf("Enter the number of rows (between 1 and 100): ");
scanf("%d", &r);
printf("Enter the number of columns (between 1 and 100): ");
scanf("%d", &c);

printf("\nEnter elements of 1st matrix:\n");
for (i = 0; i < r; ++i)
   for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &a[i][j]);
   }

printf("Enter elements of 2nd matrix:\n");
for (i = 0; i < r; ++i)
   for (j = 0; j < c; ++j) {
      printf("Enter element a%d%d: ", i + 1, j + 1);
      scanf("%d", &b[i][j]);
   }

// adding two matrices
for (i = 0; i < r; ++i)
   for (j = 0; j < c; ++j) {
      sum[i][j] = a[i][j] + b[i][j];
   }

// printing the result
printf("\nSum of two matrices: \n");
for (i = 0; i < r; ++i)
   for (j = 0; j < c; ++j) {
      printf("%d   ", sum[i][j]);
```

```
            if (j == c - 1) {
                printf("\n\n");
            }
        }

    return 0;
}
```

5.explain matrix transpose program

```
#include <stdio.h>
int main() {
    int a[10][10], transpose[10][10], r, c, i, j;
    printf("Enter rows and columns: ");
    scanf("%d %d", &r, &c);

    // Assigning elements to the matrix
    printf("\nEnter matrix elements:\n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("Enter element a%d%d: ", i + 1, j + 1);
            scanf("%d", &a[i][j]);
        }

    // Displaying the matrix a[][]
    printf("\nEntered matrix: \n");
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            printf("%d  ", a[i][j]);
            if (j == c - 1)
                printf("\n");
        }

    // Finding the transpose of matrix a
    for (i = 0; i < r; ++i)
        for (j = 0; j < c; ++j) {
            transpose[j][i] = a[i][j];
        }
```

6.Explain selection sort

selection Sort in C

Selection sort is another algorithm that is used for sorting. This sorting algorithm, iterates through the array and finds the smallest number in the array and swaps it with the first element if it is smaller than the first element. Next, it goes on to the second element and so on until all elements are sorted.

Example of Selection Sort

Consider the array:

[10,5,2,1]

The first element is 10. The next part we must find the smallest number from the remaining array. The smallest number from 5 2 and 1 is 1. So, we replace 10 by 1.

The new array is [1,5,2,10] Again, this process is repeated.

Finally, we get the sorted array as [1,2,5,10].

Let us continue with this article on Selection Sort in C and see how the algorithm works,

Algorithm for Selection Sort:

Step 1 − Set min to the first location

Step 2 − Search the minimum element in the array

Step 3 – swap the first location with the minimum value in the array

Step 4 – assign the second element as min.

Step 5 − Repeat the process until we get a sorted array.

Let us take a look at the code for the the programmatic implementation,

Code for Selection Sort:

```
1   #include <stdio.h>
2   int main()
3   {
4   int a[100], n, i, j, position, swap;
5   printf("Enter number of elementsn");
6   scanf("%d", &n);
7   printf("Enter %d Numbersn", n);
8   for (i = 0; i < n; i++)
9   scanf("%d", &a[i]);
10  for(i = 0; i < n - 1; i++)
11  {
12  position=i;
13  for(j = i + 1; j < n; j++)
14  {
15  if(a[position] > a[j])
16  position=j;
17  }
18  if(position != i)
19  {
20  swap=a[i];
21  a[i]=a[position];
22  a[position=swap;
23  }
24  }
25  printf("Sorted Array:n");
26  for(i = 0; i < n; i++)
27  printf("%dn", a[i]);
28  return 0;
29  }
```

7.Explain Linear search in C

A linear search, also known as a sequential search, is a method of finding an element within a list. It checks each element of the list sequentially until a match is found or the whole list has been searched.

A simple approach to implement a linear search is

- Begin with the leftmost element of arr[] and one by one compare x with each element.
- If x matches with an element then return the index.
- If x does not match with any of the elements then return -1.

Implementing Linear Search in C

```
#include<stdio.h>

int main()
{
    int a[20],i,x,n;
    printf("How many elements?");
    scanf("%d",&n);

    printf("Enter array elements:n");
    for(i=0;i<n;++i)
        scanf("%d",&a[i]);

    printf("nEnter element to search:");
    scanf("%d",&x);

    for(i=0;i<n;++i)
        if(a[i]==x)
            break;

    if(i<n)
        printf("Element found at index %d",i);
    else
        printf("Element not found");

    return 0;
}
```

8.Explain binary search in C

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
Example :

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

Recursive implementation of Binary Search

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
   if (r >= l) {
      int mid = l + (r - l) / 2;

      // If the element is present at the middle
      // itself
      if (arr[mid] == x)
         return mid;

      // If element is smaller than mid, then
      // it can only be present in left subarray
      if (arr[mid] > x)
         return binarySearch(arr, l, mid - 1, x);

      // Else the element can only be present
      // in right subarray
      return binarySearch(arr, mid + 1, r, x);
   }

   // We reach here when element is not
   // present in array
   return -1;
}

int main(void)
{
   int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

Output :

```
Element is present at index 3
```

Iterative implementation of Binary Search

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;

// A iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

Output :

```
Element is present at index 3
```

Time Complexity:

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

The above recurrence can be solved either using Recurrence T ree method or Master method. It falls in case II of Master Method and solution of the recurrence is .

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

Unit III Functions and pointers

PART A

1. **What is a function?**
   - ✓ Function is a set of instructions
   - ✓ Self contained block
   - ✓ Performs a specific task
   - ✓ Used to avoid redundancy of code

2. **What is the need for functions?**
   - ✓ To reduce the complexity of large programs
   - ✓ To increase the readability
   - ✓ To achieve reusability
   - ✓ To avoid redundancy of code
   - ✓ To save Memory

3. **What are the uses of pointer?**
   - ✓ Saves Memory Space
   - ✓ Used for dynamic memory allocation
   - ✓ Faster execution
   - ✓ Used to pass array of values to a function as a single argument

4. **Define typedef .**
   - ✓ The typedef keyword enables the programmer to create a new data type name by using an existing data type.
   - ✓ By using typedef, no new data is created, rather an alternate name is given to a known data type.

   Syntax: typedef existing_data_type new_data_type;

5. **What is an Address operator & Indirection operator?**

- ✓ Address operator: & -used to assign the address to a pointer variable, Referencing operator

- ✓ Indirection operator : *- Dereferencing operator is used to access the value at the pointer variable

Ex: int a=5; int

*p=&a;

printf("%d",*(p));

6. Compare actual parameter & formal argument

Actual argument: Specified in the function call statement. Used to supply the input values to the function either by copy or reference

Formal argument: Specified in the function definition statement. It takes either copy or address of the actual arguments

7. How is pointer arithmetic done?

Pointer Arithmetic:

Valid operation

- ✓ Pointer can be added with a constant
- ✓ Pointer can be subtracted with a Constant
- ✓ Pointer can be Incremented or Decremented Not

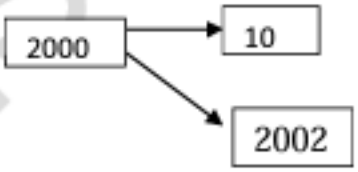
Valid

- ✓ Two pointers can not be added,subtracted,multiplied or divided

Ex: int a=10

int *p=&a;

p=p+1;



- ✓ The pointer holds the address 2000. This value is added with 1.
- ✓ The data type size of the constant is added with the address. p=

2000+(2*1)=2002

8. List out any 4 math functions

pow(x,y) : used to find power of value of x^y .Returns a double value log10(x)

: used to find natural logarithmic value of x

sqrt(x) : used to find square root vale of x

sin(x) : returns sin value of x

9. What is a function prototype?

- ✓ Function prototype is a function declaration statement. **Syntax** : return_type function_name(parameters_list) **Example:** int factorial(int);

10. Differentiate call by value and call by reference.

Call by value: The values of the variables are passed by the calling function to the called function.

Call by reference: The addresses of the variables are passed by the calling function to the called function.

11. List the header files in 'C' language.

- ✓ <stdio.h> contains standard I/O functions
- ✓ <ctype.h> contains character handling functions
- ✓ <stdlib.h> contains general utility functions
- ✓ <string.h> contains string manipulation functions
- ✓ <math.h> contains mathematical functions
- ✓ <time.h> contains time manipulation functions

12. What are the steps in writing a function in a program?

Function Declaration (Prototype declaration):

- ✓ Every user-defined functions has to be declared before the main().

Function Callings:

- ✓ The user-defined functions can be called inside any functions like main(), userdefined function, etc.

Function Definition:

- ✓ The function definition block is used to define the user-defined functions with statements.

13. State the advantages of user defined functions over pre-defined function.

- ✓ A user defined function allows the programmer to define the exact function of the module as per requirement. This may not be the case with predefined function. It may or may not serve the desired purpose completely.
- ✓ A user defined function gives flexibility to the programmer to use optimal programming instructions, which is not possible in predefined function.

14. Write the syntax for pointers to structure.

```
Struct S

{

char     datatype1;
int      datatype2;
float datatype3;

};

Struct S *sptr //sptr ia pointer to structure S
```

15. Write the advantages and disadvantages of recursion.

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion.

Recursions use

more memory and are generally slow. Instead, you can use loop.

16. What is meant by Recursive function?

- ✓ If a function calls itself again and again, then that function is called Recursive function.

Example:

```
void recursion()

{

recursion(); /* function calls itself */

}

int main()

{

recursion();

}
```

17. Is it better to use a macro or a function?

- ✓ Macros are more efficient (and faster) than function, because their corresponding code is inserted directly at the point where the macro is called.
- ✓ There is no overhead involved in using a macro like there is in placing a call to a

function. However, macros are generally small and cannot handle large, complex coding constructs.

- ✓ In cases where large, complex constructs are to handled, functions are more suited, additionally; macros are expanded inline, which means that the code is replicated for each occurrence of a macro.

18. List any five-library functions related to mathematical functions.

- ✓ ceil(x)
- ✓ sqrt(x)
- ✓ log(x)
- ✓ pow(x,y)
- ✓ sin(x)

19. What are the functions supports for dynamic memory allocation?

- ✓ The functions supports for dynamic memory allocation are,
  - ✓ malloc()
  - ✓ realloc()
  - ✓ calloc()
  - ✓ free()

20. What is linked list?

- ✓ A linked list is a collection of data elements called nodes in which the linear
- ✓ representation is given by links from one node to the next node.

PART B

1. What is a function in C? Explain the steps in writing a function in C program with Example.

Ans: A function is a block of statements that performs a specific task.

Syntax:

```
return_type function_name (argument list)
{
    Set of statements – Block of code
}
```

```
#include <stdio.h>
int addition(int num1, int num2)
{
int sum;
sum = num1+num2;
return sum;
}

int main()
{
int var1, var2;
printf("Enter number 1: ");
scanf("%d",&var1);
```

```
printf("Enter number 2: ");
scanf("%d",&var2);
int res = addition(var1, var2);
printf ("Output: %d", res);

return 0;
}
```

2. Classify the function prototypes with example C program for each.

A function prototype is a function declaration that specifies the data types of its arguments in the parameter list. The compiler uses the information in a function prototype to ensure that the corresponding function definition and all corresponding function declarations and calls within the scope of the prototype contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

Prototypes are syntactically distinguished from the old style of function declaration. The two styles can be mixed for any single function, but this is not recommended. The following is a comparison of the old and the prototype styles of declaration:

Old style:

- Functions can be declared implicitly by their appearance in a call.
- Arguments to functions undergo the default conversions before the call.
- The number and type of arguments are not checked.

Prototype style:

- Functions are declared explicitly with a prototype before they are called. Multiple declarations must be compatible; parameter types must agree exactly.
- Arguments to functions are converted to the declared types of the parameters.
- The number and type of arguments are checked against the prototype and must agree with or be convertible to the declared types. Empty parameter lists are designated using the void keyword.
- Ellipses are used in the parameter list of a prototype to indicate that a variable number of parameters are expected.

Prototype Syntax

A function prototype has the following syntax:

function-prototype-declaration:
declaration-specifiers(opt) declarator;

The *declarator* includes a parameter type list, which can consist of a single parameter of type void . In its simplest form, a function prototype declaration might have the following format:

storage_class(opt) return_type(opt) function_name (
type(1) parameter(1), ..., type(n) parameter(n));

Consider the following function definition:

```
char  function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

The corresponding prototype declaration for this function is:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
```

A prototype is identical to the header of its corresponding function definition specified in the prototype style, with the addition of a terminating semicolon (;) or comma (,), as appropriate (depending on whether the prototype is declared alone or in a multiple declaration).

Function prototypes need not use the same parameter identifiers as in the corresponding function definition because identifiers in a prototype have scope only within the identifier list. Moreover, the identifiers themselves need not be specified in the prototype declaration; only the types are required.

For example, the following prototype declarations are equivalent:

```
char  function_name( int lower, int *upper, char (*func)(), double y );
char  function_name( int a, int *b, char (*c)(), double d );
char  function_name( int, int *, char (*)(), double );
```

Though not required, identifiers should be included in prototypes to improve program clarity and increase the type-checking capability of the compiler.

Variable-length argument lists are specified in function prototypes with ellipses. At least one parameter must precede the ellipses. For example:

```
char  function_name( int lower, ... );
```

Data-type specifications cannot be omitted from a function prototype.

3. What is recursion? Write a C program to find the sum of the digits, to find the factorial of a number and binary search using recursion.

Ans: The process in which a function calls itself directly or indirectly is called **recursion** and the corresponding function is called as **recursive** function.

Factorial:

```
#include<stdio.h>
long int multiplyNumbers(int n);
int main() {
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   printf("Factorial of %d = %ld", n, multiplyNumbers(n));
   return 0;
}
```

```
long int multiplyNumbers(int n) {
   if (n>=1)
      return n*multiplyNumbers(n-1);
   else
      return 1;
}
```

Binary search:

```
#include <stdio.h>
void binary_search(int [], int, int, int);
 void bubble_sort(int [], int);

 int main()
 {
    int key, size, i;
    int list[25];

    printf("Enter size of a list: ");
    scanf("%d", &size);
    printf("Enter elements\n");
    for(i = 0; i < size; i++)
    {
       scanf("%d",&list[i]);
    }
    bubble_sort(list, size);
    printf("\n");
    printf("Enter key to search\n");
    scanf("%d", &key);
    binary_search(list, 0, size, key);

 }

 void bubble_sort(int list[], int size)
 {
    int temp, i, j;
    for (i = 0; i < size; i++)
    {
       for (j = i; j < size; j++)
       {
          if (list[i] > list[j])
          {
             temp = list[i];
             list[i] = list[j];
             list[j] = temp;
          }
       }
```

```
    }
}

void binary_search(int list[], int lo, int hi, int key)
{
    int mid;

    if (lo > hi)
    {
        printf("Key not found\n");
        return;
    }
    mid = (lo + hi) / 2;
    if (list[mid] == key)
    {
      printf("Key found\n");
   }
    else if (list[mid] > key)
    {
        binary_search(list, lo, mid - 1, key);
    }
    else if (list[mid] < key)
   {
        binary_search(list, mid + 1, hi, key);
}
}
```

4. Write a C program to design the scientific calculator using built-in functions.

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159265
float sine(float  x);
float cosine(float x);
float tangent(float x);
float sineh(float x);
float cosineh(float x);
float tangenth(float x);
float logten(float x);
float squareroot(float x);
float exponent(float x);
float power(float x,float y);
int main()
{
   int x,y,n,answer;
```

```
    printf("What do you want to do?\n");
    printf("1.sin 2.cos 3. tan 4. sinh 5.cosh 6.tanh 7.1og10 8. square root. 9.exponent 10.power.");
    scanf ("%d",&n);
    if (n<9 && n>0)
    {
       printf("\n What is x? ");
       scanf("%f",&x);
       switch (n)
       {
          case 1: answer = sine(x);       break;
          case 2: answer = cosine(x);     break;
          case 3: answer = tangent(x);    break;
          case 4: answer = sineh(x);      break;
          case 5: answer = cosineh(x);    break;
          case 6: answer = tangenth(x);   break;
          case 7: answer = logten(x);     break;
          case 8: answer = squareroot(x); break;
          case 9: answer = exponent(x);   break;
       }
    }
    if (n==10)
    {
       printf("What is x and y?\n");
       scanf("%f%f",&x,&y);
       answer = power(x,y);
    }
    if (n>0 && n<11)
       printf("%f",answer);
    else
       printf("Wrong input.\n");
    return 0;
}
float sine(float x)
{
    return (sin (x*PI/180));
}
float cosine(float x)
{
    return (cos (x*PI/180));
}
float tangent(float x)
{
    return (tan(x*PI/180));
}
float sineh(float x)
{
    return (sinh(x));
```

```
}
float cosineh(float x)
{
   return (sinh(x));
}
float tangenth(float x)
{
   return (sinh(x));
}
float logten(float x)
{
   return (log10(x));
}
float squareroot(float x)
{
   return (sqrt(x));
}
float exponent(float x)
{
   return(exp(x));
}
float power(float x, float y)
{
   return (pow(x,y));
}
```

5. Explain about pointers and write the use of pointers in arrays with suitable example.

Ans: Pointers in C language is a variable that stores/points the address of another variable. A Pointer in C is used to allocate memory dynamically i.e. at run time. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Pointer Syntax : data_type *var_name; Example : int *p; char *p;

Where, * is used to denote that "p" is pointer variable and not a normal variable.

```
#include <stdio.h>

int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d", *p);
      p++;
```

```
    }

    return 0;
}
```

6. Explain the concept of pass by value and pass by reference. Write a C program to swap the content of two variables using pass by reference.

` **Pass by value**: In this approach we pass copy of actual variables in function as a parameter. Hence any modification on parameters inside the function will not reflect in the actual variable. For example:

```
#include<stdio.h>
int main(){
   int a=5,b=10;
   swap(a,b);
   printf("%d     %d",a,b);
   return 0;
}
void swap(int a,int b){
   int temp;
   temp =a;
   a=b;
   b=temp;
}
```

Output: 5 10

(b)**Pass by reference**: In this approach we pass memory address actual variables in function as a parameter. Hence any modification on parameters inside the function will reflect in the actual variable. For example:

```
#incude<stdio.h>
```

```
int main(){
   int a=5,b=10;
   swap(&a,&b);
   printf("%d %d",a,b);
   return 0;
}
void swap(int *a,int *b){
   int  *temp;
   *temp =*a;
   *a=*b;
   *b=*temp;
}
```

Output: 10 5

UNIT IV

STRUCTRES

1. Compare arrays and structures.

Comparison of arrays and structures is as follows.

Arrays	**Structures**
An array is a collection of data items of same data type. Arrays can only be declared.	A structure is a collection of data items of different data types. Structures can be declared and defined.
There is no keyword for arrays.	The keyword for structures is struct.
An array cannot have bit fields.	A structure may contain bit fields.
An array name represents the address of the starting element.	A structure name is known as tag. It is a Shorthand notation of the declaration.

2. Compare structures and unions.

Structure	**Union**

Every member has its own memory.	All members use the same memory.
The keyword used is struct.	The keyword used is union.
All members occupy separate memory location, hence different interpretations of the same memory location are not possible. Consumes more space compared to union.	Different interpretations for the same memory location are possible. Conservation of memory is possible

3. Define Structure in C.

C Structure is a collection of different data types which are grouped together and each element in a C structure is called member.

If you want to access structure members in C, structure variable should be declared.

Many structure variables can be declared for same structure and memory will be allocated for each separately.

It is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.

4. What you meant by structure definition?

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition. typedef struct { char

```
    name[64];
    char course[128];
    int age;
    int year;
} student;
```

This defines a new type student variables of type student can be declared as follows. student st_rec;

5. How to Declare a members in Structure?

A **struct** in C programming language is a structured (record) type[1] that aggregates a fixed set of labeled objects, possibly of different types, into a single object. The syntax for a struct declaration in C is:

```
struct tag_name
{
  type    attribute;    type
  attribute2;
  /* ... */
};
```

6. What is meant by Union in C?

A **union** is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

7. How to define a union in C.

To define a union, you must use the **union** statement in very similar was as you did while defining structure. The union statement defines a new data type, with more than one member for your program. The format of the union statement is as follows:

```
union [union tag]
{
  member      definition;
  member definition;
  ...
  member definition;
} [one or more union variables];
```

8. What are storage classes?

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

9. What are the storage classes available in C?

There are following storage classes which can be used in a C Program

1. auto
2. register
3. static
4. extern

10. What is register storage in storage class?

Register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int Miles;
}
```

11. What is static storage class?

Static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```
static int Count; int Road;
{
printf("%d\n", Road);
}
```

12. Define Auto storage class in C.

auto is the default storage class for all local variables.

```
{
int Count; auto int Month;
}
```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

13. Define Macro in C.

A macro definition is independent of block structure, and is in effect from the #define directive that defines it until either a corresponding #undef directive or the end of the compilation unit is encountered.

Its format is: #define identifier replacement

Example:

```
#define
TABLE_SIZ
E                100
int
table1[TAB
LE_SIZ E];       int
table2[TAB
LE_SIZ E];
```

14. What is Line control? Line control (#line)

When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

#line number "filename"

Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

15. What is the use of 'typedef"?

It is used to create a new data using the existing type.

Syntax: typedef data type name;

Example:

typedef int hours: hours hrs;/* Now, hours can be used as new datatype */

16. Write the syntax for pointers to structure.

```
Struct  S

{
char datatype1; int
datatype2; float
datatype3;
};

Struct S *sptr; //sptr ia pointer to structure S
```

17. Define self referential data structure

A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind. A chain of such structures can thus be expressed as follows.

```
struct name {
        member 1;
        member 2;
        . . .
        struct name *pointer;
};
```

The above illustrated structure prototype describes one node that comprises of two logical segments. One of them stores data/information and the other one is a pointer indicating where the next component can be found. .Several such inter-connected nodes create a chain of structures.

18. What is singly linked list?

A linear linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as *startpointer*) is used. The end of the list is indicated by a *NULL*pointer. In order to create a linked list of integers, we define each of its element (referred as *node*) using the following declaration.

```
struct node_type {

        int data;

        struct node_type *next;


};

struct node_type *start = NULL;
```

Note: The second member points to a node of same type. A linear linked list illustration:

19. What are the various dynamic memory allocation functions?

- malloc() - Used to allocate blocks of memory in required size of bytes.
- free () - Used to release previously allocated memory space.
- calloc() - Used to allocate memory space for an array of elements.
- realloac() - Used to modify the size of the previously allocated memory space.

20. How to create a node in linked list?

The basic thing is to create a node. It can be done by using the malloc function.

start = (node*) malloc(sizeof(node))

This statement creates the starting node of list. A block of memory whose size is equal to the sizeof(node) is allocated to the node pointer start. Typecasting is used because otherwise malloc will return pointer to character.

PART-B

1. What is a structure? Create a structure with data members of various types and declare two structure variables. Write a program to read data into these and print the same. Justify the need for structured data type.

It is the collection of dissimilar data types or heterogenous data types grouped together. It means the data types may or may not be of same type.

Structure

declaration- struct

tagname

{

Data type

member1; Data

type member2;

Data type

member3;

……

……

Data type member n;

}

;

struct

{

Data type member1;

Data type member2;

Data type member3;

……

……

Data type member n;

}

;

struct tagname

{

struct element 1;

struct element 2;

struct element 3;

……

……

struct element n;

};

Structure variable declaration;

```
struct student

{

int age; char name[20];

 char branch[20];

}; struct student s;
```

Initialization of structure variable-

Like primary variables structure variables can also be initialized when they are declared. Structure templates can be defined locally or globally. If it is local it can be used within that function. If it is global it can be used by all other functions of the program.

We cant initialize structure members while defining the structure

```
struct student

    {

    int age=20;

    char
name[20]=”sona”;

    }s1;
```

The above is **invalid.**

A structure can be initialized as

```
struct student

    {
```

int age,roll;

char name[20];

} struct student

s1={16,101,"sona"}; struct student

s2={17,102,"rupa"};

If initialiser is less than no.of structure variable, automatically rest values are taken as zero.

Accessing structure elements-

Dot operator is used to access the structure elements. Its associativety is from left to right.

structure variable ; s1.name[];

s1.roll; s1.age;

Elements of structure are stored in contiguous memory locations. Value of structure variable can be assigned to another structure variable of same type using assignment operator.

Example:

#include<stdio.h> #include<conio.h> void main()

{

int roll, age; char branch;

} s1,s2;

printf("\n enter roll, age, branch=");

scanf("%d %d %c", &s1.roll, &s1.age, &s1.branch); s2.roll=s1.roll;

printf(" students details=\n");

printf("%d %d %c", s1.roll, s1.age, s1.branch); printf("%d", s2.roll);

```
}
```

Unary, relational, arithmetic, bitwise operators are not allowed within structure variables.

2. **Write a C program to store the employee information using structure and search a particular employee using Employee number.**

```
/*C program to read and print employee's record using structure*/

#include <stdio.h>

/*structure declaration*/
struct employee{
   char    name[30];
   int     empId;
   float   salary;
};

int main()
{
   /*declare structure variable*/
   struct employee emp;

   /*read employee details*/
   printf("\nEnter details :\n");
   printf("Name ?:");          gets(emp.name);
   printf("ID ?:");           scanf("%d",&emp.empId);
   printf("Salary ?:");       scanf("%f",&emp.salary);

   /*print employee details*/
   printf("\nEntered detail is:");
   printf("Name: %s"   ,emp.name);
   printf("Id: %d"     ,emp.empId);
   printf("Salary: %f\n",emp.salary);
   return 0;}
}
```

3. **Define and declare a nested structure to store date, which including day, month and year.**

Nesting of structure within itself is not valid. Nesting of structure can be extended to any level.

```
struct time

{

int hr,min;

};

struct day
```

```
{
int date,month; struct time t1;
};
struct student
{
char        nm[20];
struct day d;
}stud1, stud2, stud3;
```

4. Explain about array of structures and pointers in structures with example program.

Array of structures

When database of any element is used in huge amount, we prefer Array of structures.

Example: suppose we want to maintain data base of 200 students, Array of structures is used.

```
#include<stdio.h>
#include<string.h>
struct student
{
char  name[30];  char
branch[25]; int roll;
};
void main()
{
struct student s[200]; int i;
s[i].roll=i+1;
```

```
printf("\nEnter information of students:");
for(i=0;i<200;i++)
{
printf("\nEnter the roll no:%d\n",s[i].roll);
printf("\nEnter the name:"); scanf("%s",s[i].name);
printf("\nEnter the branch:");
scanf("%s",s[i].branch); printf("\n");
}
printf("\nDisplaying information of students:\n\n");
for(i=0;i<200;i++)
{
printf("\n\nInformation for roll no%d:\n",i+1);
printf("\nName:");
puts(s[i].name);
printf("\nBranch:");
puts(s[i].branch);
}
}
```

In Array of structures each element of array is of structure type as in above example.

Array within structures

```
struct student
{
```

```
char name[30];

int roll,age,marks[5];

}; struct student s[200];
```

We can also initialize using same syntax as in array.

5. Write a C program to create mark sheet for students using self referential structure.

```
 #include<stdio.h>
#include<conio.h>
struct mark_sheet{
        char name[20];
        long int rollno;
        int marks[10];
        int total;
        float average;
        char rem[10];
        char cl[20];
        }students[100];

int main(){
        int a,b,n,flag=1;
        char ch;
        clrscr();
        printf("How many students : \n");
        scanf("%d",&n);
        for(a=1;a<=n;++a){
```

```
clrscr();
printf("\n\nEnter the details of %d students : ", n-a+1);
printf("\n\nEnter student  %d  Name : ", a);
scanf("%s", students[a].name);
printf("\n\nEnter student %d Roll Number : ", a);
scanf("%ld", &students[a].rollno);
students[a].total=0;
for(b=1;b<=5;++b){
printf("\n\nEnter the mark of subject-%d :  ", b);
scanf("%d", &students[a].marks[b]);
students[a].total += students[a].marks[b];
if(students[a].marks[b]<40)
flag=0;
}
students[a].average = (float)(students[a].total)/5.0;
if((students[a].average>=75)&&(flag==1))
strcpy(students[a].cl,"Distinction");
else
if((students[a].average>=60)&&(flag==1))
strcpy(students[a].cl,"First Class");
else
if((students[a].average>=50)&&(flag==1))
strcpy(students[a].cl,"Second Class");
else
if((students[a].average>=40)&&(flag==1))
strcpy(students[a].cl,"Third Class");
if(flag==1)
```

```
strcpy(students[a].rem,"Pass");
else
strcpy(students[a].rem,"Fail");
flag=1;
}
for(a=1;a<=n;++a){
clrscr();
printf("\n\n\t\t\t\tMark Sheet\n");
printf("\nName of Student : %s", students[a].name);
printf("\t\t\t\t Roll No : %ld", students[a].rollno);
printf("\n---------------------------------------------------------------------");
for(b=1;b<=5;b++){
printf("\n\n\t Subject %d \t\t :\t %d", b, students[a].marks[b]);
}
printf("\n\n---------------------------------------------------------------------\n");
printf("\n\n Totl Marks : %d", students[a].total);
printf("\t\t\t\t Average Marks : %5.2f", students[a].average);
printf("\n\n Class : %s", students[a].cl);
printf("\t\t\t\t\t Status : %s", students[a].rem);
printf("\n\n\n\t\t\t\t Press Y for continue . . . ");
ch = getche();
if((ch=="y")||(ch=="Y"))
continue;
}
return(0);
}
```

6. Discuss about dynamic memory allocation with suitable example C program.

Dynamic memory Allocation

The process of allocating memory at the time of execution or at the runtime, is called dynamic memory location.

Two types of problem may occur in static memory allocation.

If number of values to be stored is less than the size of memory, there would be wastage of memory.

If we would want to store more values by increase in size during the execution on assigned size then it fails.

Allocation and release of memory space can be done with the help of some library function called dynamic memory allocation function. These library function are called as **dynamic memory allocation function.** These library function prototype are found in the header file, "alloc.h" where it has defined.

Function take memory from memory area is called heap and release when not required.

Pointer has important role in the dynamic memory allocation to allocate memory.

malloc():

This function use to allocate memory during run time, its declaration is void*malloc(size);

malloc ()

returns the pointer to the 1st byte and allocate memory, and its return type is void, which can be type cast such as:

int *p=(datatype*)malloc(size)

If memory location is successful, it returns the address of the memory chunk that was allocated and it returns null on unsuccessful and from the above declaration a pointer of type **(datatype)** and size in byte.

And **datatype** pointer used to typecast the pointer returned by malloc and this typecasting is necessary since, malloc() by default returns a pointer to void.

Example int*p=(int*)malloc(10);

So, from the above pointer p, allocated IO contigious memory space address of 1st byte and is stored in the variable.

We can also use, the size of operator to specify the the size, such as

p=(int)malloc(5*size of int) Here, 5 is the no. of data.

Moreover , it returns null, if no sufficient memory available , we should always check the malloc return such as, **if(p==null)**

printf("not sufficient memory");

Example:

```
/*calculate the average of mark*/ void
main()
{
int n , avg,i,*p,sum=0;
    printf("enter the no. of marks ");
    scanf("%d",&n);
    p=(int *)malloc(n*size(int));
    if(p==null)
    printf("not sufficient"); exit();
    }
    for(i=0;i<n;i++)
    scanf("%d",(p+i));
    for(i=0;i<n;i++)
    Printf("%d",*(p+i));
    sum=sum+*p;
```

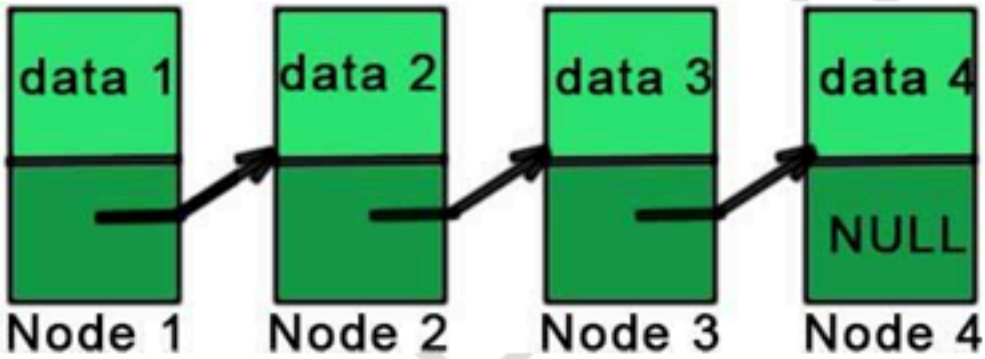
```
avg=sum/n;

printf("avg=%d",avg);
```

7. Explain about singly linked list with suitable example C program.

Linked list is one of the most important data structures. We often face situations, where the data is dynamic in nature and number of data can't be predicted or the number of data keeps changing during program execution. Linked lists are very useful in this type of situations.

The implementation of a linked list in C is done using pointers

A linked list is made up of many nodes which are connected in nature. Every node is mainly divided into two parts, one part holds the data and the other part is connected to a different node. It is similar to the picture given below.




```
include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

struct test_struct
{
   int val;
   struct test_struct *next;
};

struct test_struct *head = NULL;
struct test_struct *curr = NULL;

struct test_struct* create_list(int val)
{
   printf("\n creating list with headnode as [%d]\n",val);
   struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
   if(NULL == ptr)
   {
      printf("\n Node creation failed \n");
      return NULL;
   }
   ptr->val = val;
   ptr->next = NULL;

   head = curr = ptr;
```

```
    return ptr;
}

struct test_struct* add_to_list(int val, bool add_to_end)
{
    if(NULL == head)
    {
        return (create_list(val));
    }

    if(add_to_end)
        printf("\n Adding node to end of list with value [%d]\n",val);
    else
        printf("\n Adding node to beginning of list with value [%d]\n",val);

    struct test_struct *ptr = (struct test_struct*)malloc(sizeof(struct test_struct));
    if(NULL == ptr)
    {
        printf("\n Node creation failed \n");
        return NULL;
    }
    ptr->val = val;
    ptr->next = NULL;

    if(add_to_end)
    {
        curr->next = ptr;
        curr = ptr;
    }
    else
    {
        ptr->next = head;
        head = ptr;
    }
    return ptr;
}

struct test_struct* search_in_list(int val, struct test_struct **prev)
{
    struct test_struct *ptr = head;
    struct test_struct *tmp = NULL;
    bool found = false;

    printf("\n Searching the list for value [%d] \n",val);

    while(ptr != NULL)
    {
        if(ptr->val == val)
        {
            found = true;
            break;
        }
```

```
        else
        {
            tmp = ptr;
            ptr = ptr->next;
        }
    }

    if(true == found)
    {
        if(prev)
            *prev = tmp;
        return ptr;
    }
    else
    {
        return NULL;
    }
}

int delete_from_list(int val)
{
    struct test_struct *prev = NULL;
    struct test_struct *del = NULL;

    printf("\n Deleting value [%d] from list\n",val);

    del = search_in_list(val,&prev);
    if(del == NULL)
    {
        return -1;
    }
    else
    {
        if(prev != NULL)
            prev->next = del->next;

        if(del == curr)
        {
            curr = prev;
        }
        else if(del == head)
        {
            head = del->next;
        }
    }

    free(del);
    del = NULL;

    return 0;
}
```

```
void print_list(void)
{
   struct test_struct *ptr = head;

   printf("\n -------Printing list Start------- \n");
   while(ptr != NULL)
   {
      printf("\n [%d] \n",ptr->val);
      ptr = ptr->next;
   }
   printf("\n -------Printing list End------- \n");

   return;
}

int main(void)
{
   int i = 0, ret = 0;
   struct test_struct *ptr = NULL;

   print_list();

   for(i = 5; i<10; i++)
      add_to_list(i,true);

   print_list();

   for(i = 4; i>0; i--)
      add_to_list(i,false);

   print_list();

   for(i = 1; i<10; i += 4)
   {
      ptr = search_in_list(i, NULL);
      if(NULL == ptr)
      {
         printf("\n Search [val = %d] failed, no such element found\n",i);
      }
      else
      {
         printf("\n Search passed [val = %d]\n",ptr->val);
      }

      print_list();

      ret = delete_from_list(i);
      if(ret != 0)
      {
         printf("\n delete [val = %d] failed, no such element found\n",i);
      }
      else
```

```
        {
            printf("\n delete [val = %d]  passed \n",i);
        }

        print_list();
    }

    return 0;
}
```

UNIT V
FILE PROCESSING

1. Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

2. Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

Text files

Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

3. Enlist the File Operations.

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

FILE *fptr;

4. How to open a file?

Opening a file is performed using the library function in the **"stdio.h"** header file: fopen().

The syntax for opening a file in standard I/O is:

ptr = fopen("fileopen","mode")

5. List the opening modes in standard I/O

File Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.

r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

6. How to close a file?

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function fclose().

fclose(fptr); //fptr is the file pointer associated with file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

7. What are two main ways a file can be organized?

1. **Sequential Access** — The data are placed in the file in a sequence like beads on a string. Data are processed in sequence, one after another. To reach a particular item of data, all the data that proceeds it first must be read.
2. **Random Access** — The data are placed into the file by going directly to the location in the file assigned to each data item. Data are processed in any order. A particular item of data can be reached by going directly to it, without looking at any other data.

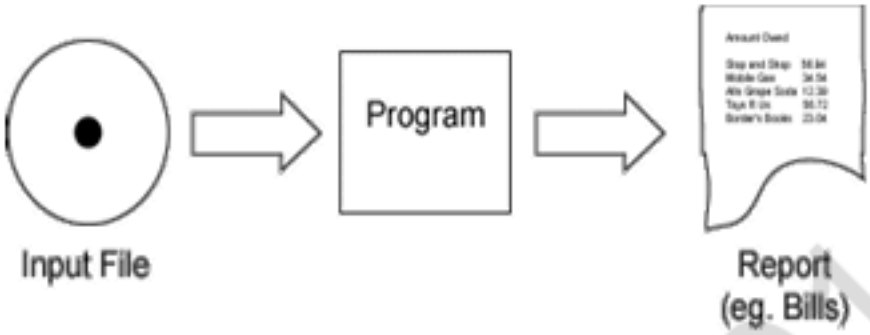
8. What is file?

A file is a semi-permanent, named collection of data. A File is usually stored on magnetic media, such as a hard disk or magnetic tape.

Semi-permanent means that data saved in files stays safe until it is deleted or modified.

Named means that a particular collection of data on a disk has a name, like mydata.dat and access to the collection is done by using its name.

9. State Report Generation.

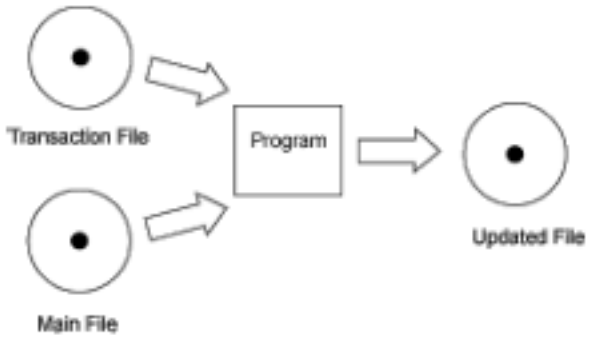


A very common data processing operation is **report generation**. This is when the data in a file (or files) is processed by a program to generate a report of some kind. The report might be a summary of the financial state of a corporation, or a series of bills for its customers, or a series of checks to be mailed to its employees.

Large corporations have very large databases kept on many high capacity disks and processed by programs running on large computers called **mainframe**

10. State Transaction Processing.

As data flows into an organization, the files that keep track of the data must be updated. For example, data flows into a bank from other banks and ATM machines. Often this data is gathered as it occurs into a **transaction file**. Periodically (often overnight) the data in the transaction file is used to update the main file of data.



The picture shows the main file and the transaction file as input to a program. The output is a new, updated main file. The previous version of the file can be kept as backup.

Management information science is the field that studies how to organize and manage the information of a corporation using computers and files. Usually the business school of a university has a management information science department.

11. List the Types of Files.

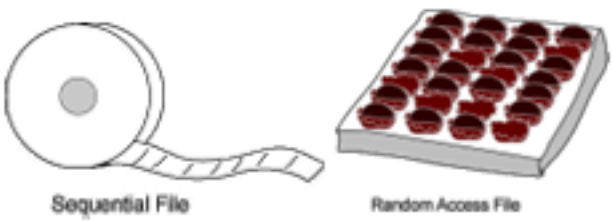
There are two main ways a file can be organized:

1. **Sequential Access** — The data are placed in the file in a sequence like beads on a string. Data are processed in sequence, one after another. To reach a particular item of data, all the data that proceeds it first must be read.
2. **Random Access** — The data are placed into the file by going directly to the location in the file assigned to each data item. Data are processed in any order. A particular item of data can be reached by going directly to it, without looking at any other data.

A sequential file works like a reel of tape. (In fact, sequential files are often stored on reels of magnetic tape.) Data in a sequential file are processed in order, starting with the first item, the processing the second, then the third and so on. To reach data in the middle of the file you must go through all the data that preceeds it.

A random access file works like a sampler box of chocolate. You can access a particular item by going directly to it. Data in a random acess file may be accessed in any order. To read data in the middle of the file you can go directly to it. Random access files are sometimes

called **direct access** files.



You might think that all files should be random access. But random access files are much more difficult to imp.

12. What is command line arguments?

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the number of arguments passed, and argv[] is a pointer array which points to each argument passed to the program.

13. Write an example program for command line arguments.

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
if( argc == 2 ) {
                printf("The argument supplied is %s\n", argv[1]);
                }
     else if( argc > 2 )
{
                printf("Too many arguments supplied.\n");
                }
  else
                {
    printf("One argument expected.\n");
                }
}
```

14. Write the functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

15. Write short notes on fseek().

fseek():

This function is used for seeking the pointer position in the file at the specified byte.

Syntax: fseek(file pointer, displacement, pointer position);

Where

file pointer -----It is the pointer which points to the file.
displacement ---- It is positive or negative.This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position.This is attached with L because this is a long integer.

Pointer position:

This sets the pointer position in the file.

Value	pointer position
0	Beginning o
1	Current position
2	End of file

16. Give an example for fseek().

1) fseek(p,10L,0)
0 means pointer position is on beginning of the file,from this statement pointer position is skipped 10 bytes from the beginning of the file.

2) fseek(p,5L,1)
1 means current position of the pointer position.From this statement pointer position

is skipped 5 bytes forward from the current position.

3) fseek(p,-5L,1)

From this statement pointer position is skipped 5 bytes backward from the current position.

17. Give an example for ftell().

ftell()

This function returns the value of the current pointer position in the file.The value is count from the beginning of the file.

Syntax: ftell(fptr);

Where fptr is a file pointer.

18. Give an example for rewind().

rewind()

This function is used to move the file pointer to the beginning of the given file.

Syntax: rewind(fptr);

Where fptr is a file pointer.

19. State Block read/write.

It is useful to store the block of data into the file rather than individual elements. Each block has some fixed size, it may be of structure or of an array. It is possible that a data file has one or more structures or arrays, So it is easy to read the entire block from file or write the entire block to the file. There are two useful functions for this purpose

20. Write short notes on fwrite().

This function is used for writing an entire block to a given file.

Syntax: fwrite(ptr, size, nst, fptr);

Where ptr is a pointer which points to the arrayof struture in which data is written.

Size is the size of the structure

nst is the number of the structure

fptr is a filepointer.

Example program for fwrite():

21. Write short notes on fread().

This function is used to read an entire block from a given file.

Syntax: fread (ptr , size , nst , fptr);

Where **ptr** is a pointer which points to the array which receives structure.

Size is the size of the structure
nst is the number of the structure
fptr is a filepointer.

22. Write short notes on fprintf ().

This function is same as the printf() function but it writes the data into the file, so it has one more parameter that is the file pointer.

Syntax: fprintf(fptr, "controlcharacter",variable-names);

Where **fptr** is a file pointer

Control character specifies the type of data to be printed into file.

Variable-names hold the data to be printed into the file.

23. Write short notes on fscanf ().

This function is same as the scanf() function but this reads the data from the file, so this has one more parameter that is the file pointer.

Syntax: fscanf(fptr, "control character", &variable-names);

Where **fptr** is a file pointer

Control character specifies the type of data to be read from the file.
Address of Variable names are those that hold the data read from the file.

24. Write short notes on fscanf ().

The macro feof() is used for detecting whether the file pointer is at the end of file or not.It returns nonzero if the file pointer is at the end of the file otherwise it returns zero.

Syntax: feof(fptr);

Where fptr is a file pointer .

25. Write

short notes on

ferror().

ferror()

The macro ferror() is used for detecting whether an error occur in the file on filepointer or not.It returns the value nonzero if an error,otherwise it returns zero.

Syntax: ferror(fptr);

Where fptr is a file pointer.

PART-B

1. Explain about files and with it types of file processing.

Files: As we know that Computers are used for storing the information for a Permanent Time or the Files are used for storing the Data of the users for a Long time Period. And the files can contains any type of information means they can Store the text, any Images or Pictures or any data in any Format. So that there must be Some Mechanism those are used for Storing the information, Accessing the information and also Performing Some Operations on the files

There are Many files which have their Owen Type and own names. When we Store a File in the System, then we must have to specify the Name and the Type of File. The Name of file will be any valid Name and Type means the application with the file has linked.

So that we can say that Every File also has Some Type Means Every File belongs to Special Type of Application software's. When we Provides a Name to a File then we also specify the Extension of the File because a System will retrieve the Contents of the File into that Application Software. For Example if there is a File Which Contains Some Paintings then this will Opened into the Paint Software.
1) Ordinary Files or Simple File: Ordinary File may belong to any type of Application for example notepad, paint, C Program, Songs etc. So all the Files those are created by a user are Ordinary Files. Ordinary Files are used for Storing the information about the user Programs. With the help of Ordinary Files we can store the information which contains text, database, any image or any other type of information.
2) Directory files: The Files those are Stored into the a Particular Directory or Folder. Then these are the Directory Files. Because they belongs to a Directory and they are Stored into a Directory or Folder. For Example a Folder Name Songs which Contains Many Songs So that all the Files of Songs are known as Directory Files.
3) Special Files: The Special Files are those which are not created by the user. Or The Files those are necessary to run a System. The Files those are created by the System. Means all the Files of an Operating System or Window, are refers to Special Files. There are Many Types of Special Files, System Files, or windows Files, Input output Files. All the System Files are Stored into the System by using. sys Extension.
4) FIFO Files: The First in First Out Files are used by the System for Executing the Processes into Some Order. Means To Say the Files those are Come first, will be Executed First and the System Maintains a Order or Sequence Order. When a user Request for a Service from the System, then the Requests of the users are Arranged into Some Files and all the Requests of the System will be performed by the System by using Some Sequence Order in which they are Entered or we can say that all the files or Requests those are Received from the users will be Executed by using Some Order which is also called as First in First Out or FIFO order.

Types of File Operations

Files are not made for just reading the Contents, we can also Perform Some other operations on the Files those are Explained below As :

1) Read Operation: Meant To Read the information which is Stored into the Files.
2) Write Operation: For inserting some new Contents into a File.
3) Rename or Change the Name of File.
4) Copy the File from one Location to another.
5) Sorting or Arrange the Contents of File.
6) Move or Cut the File from One Place to Another.
7) Delete a File
8) Execute Means to Run Means File Display Output.

2. Compare sequential access and random access.

Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.

On the other hand, **Random Access** to a file means that the computer system can read or write information anywhere in the data file. This type of operation is also called "Direct Access" because the computer system knows where the data is stored (using Indexing) and hence goes "directly" and reads the data.

Sequential access has advantages when you access information in the same order all the time. Also is faster than random access.

On the other hand, random access file has the advantage that you can search through it and find the data you need more easily (using indexing for example). Random Access Memory (RAM) in computers works like that.

3. **Write a C program to read name and marks of n number of students from user and store them in a file.**

```
#include <stdio.h>
#include<conio.h>
int main(){
  char name[50];
  int marks,i,n;
  clrscr();
  printf("Enter number of students: ");
  scanf("%d",&n);
  FILE *fptr;
  fptr=(fopen("C:\\student.txt","w"));
  if(fptr==NULL){
     printf("Error!");
     exit(1);
  }
  for(i=0;i<n;++i)
  {
    printf("For student%d\nEnter name: ",i+1);
    scanf("%s",name);
    printf("Enter marks: ");
    scanf("%d",&marks);
   fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
  }
  fclose(fptr);
  getch();
  return 0;
}
```

4. **Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.**

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

```
int main(){
  char name[50];
  int marks,i,n;
  clrscr();
  printf("Enter number of students: ");
  scanf("%d",&n);
  FILE *fptr;
  fptr=(fopen("C:\\student.txt","a"));
  if(fptr==NULL){
      printf("Error!");
      exit(1);
  }
  for(i=0;i<n;++i)
  {
    printf("For student%d\nEnter name: ",i+1);
    scanf("%s",name);
    printf("Enter marks: ");
    scanf("%d",&marks);
    fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
  }
  fclose(fptr);
  getch();
  return 0;
}
```

5. Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include<stdio.h>
#include<conio.h>
struct s
{
char name[50];
int height;
};
int main(){
   struct s a[5],b[5];
   FILE *fptr;
   int i;
clrscr();
   fptr=fopen("file.txt","wb");
   for(i=0;i<5;++i)
   {
      fflush(stdin);
```

```
        printf("Enter name: ");
        gets(a[i].name);
        printf("Enter height: ");
        scanf("%d",&a[i].height);
    }
    fwrite(a,sizeof(a),1,fptr);
    fclose(fptr);
    fptr=fopen("file.txt","rb");
    fread(b,sizeof(b),1,fptr);
    for(i=0;i<5;++i)
    {   printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
    }
    fclose(fptr);
  getch();
  }
```

6. Describe command line arguments with example C program.

Command line arguments are the arguments specified after the program name in the operating system's command line, and these arguments values are passed to your program at the time of execution from your operating system. For using this concept in your program, you have to understand the complete declaration of how the main function works with this command-line argument to fetch values that earlier took no arguments with it (main() without any argument).

So you can program the main() is such a way that it can essentially accept two arguments where the first argument denotes the number of command line arguments whereas the second argument denotes the full list of every command line arguments. This is how you can code your command line argument within the parenthesis of main():

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

In the above statement, the command line arguments have been handled via main() function, and you have set the arguments where

- argc (ARGument Count) denotes the number of arguments to be passed and
- argv [ ] (ARGument Vector) denotes to a pointer array that is pointing to every argument that has been passed to your program.

```
#include <stdio.h>

int main( int argc, char *argv [] )
{
  printf(" \n Name of my Program %s \t", argv[0]);

  if( argc == 2 )
  {
```

```
        printf("\n Value given by user is: %s \t", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("\n Many values given by users.\n");
    }
    else
    {
        printf(" \n Single value expected.\n");
    }
}
```