UNIT I

PART A

1. **Compare While, Do-While and For Statement?**

While Loop: It provides a mechanism to repeat one or more statements while a particular statement is true, in while loop first the condition is tested and then the block of statement is executed. It is an entry controlled loop Structure.

Syntax:

```
Statement x;
While(condition)
{
Statement block;
}
Statement y;
```

Do-while: This loop is similar to while loop, the only difference is that in do-while loop the condition is tested at the end of the loop. It is an exit controlled loop Structure.

Syntax:

```
Statement x;
do
{
Statement block;
} While(condition);
Statement y;
```

For loop: It used the loop variable is initialized only once. With every iteration of the loop, the value of loop variable is updated and the condition is checked. If the condition is true the statement block for the loop is executed else the control jumps to immediate statement following the For loop. It is an entry controlled loop Structure.

Syntax:

```
for(initialization;condition;increment/decrement/update)
{
Statement Block;
}
Statement y;
```

2. **Differentiate between pointer to constants and constants pointers**

Constant Pointers : These type of pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). If we try to point the pointer to some other variable (or try to make the pointer store address of some other variable), then constant pointers are incapable of this.

A constant pointer is declared as : 'int *const ptr' (the location of 'const' make the pointer 'ptr' as constant pointer).

Example:

```
#include<stdio.h>
int main(void)
{
   int a[] = {10,11};
   int* const ptr = a;
   *ptr = 11;
   printf("\n value at ptr is  : %d\n",*ptr);
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   ptr++;
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   return 0;

 }
```

Pointer to Constant : These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.

A pointer to a constant is declared as : 'const int *ptr' (the location of 'const' makes the pointer 'ptr' as a pointer to constant.

Example:

```
#include<stdio.h>
int main(void)
{
   int a = 10;
   const int* ptr = &a;
   printf("\n value at ptr is  : %d\n",*ptr);
   printf("\n Address pointed by ptr  : %p\n",(unsigned int*)ptr);
   *ptr = 11;
   return 0;
}
```

3. What is Sizeof operator in C?

Sizeof is unary operator used to calculate the sizes of data types, this operator can be applied to all data types. The size of operator is used to determine the amount of memory space that the variable/expression/ data type will take.

For example,

int a=10, result

Result=sizeof (a);

Output:

result=2.

Since a is integer, it requires 2 bytes of storage space.

4. **How many storage classes does C language supports. Why we need these types of such classes? Explain each Type with an Example.**

Storage class of a variable defines the scope (visibility) and life time of variables. It is needed because it specifies how long the storage allocation will continue to exist for the function or variable. It specifies the scope of variable or function. It specify whether variable will automatically initialized to zero or in-determinate value. C supports four storage classes: automatic, register, external, and static.

Auto: The auto storage class specified is used to explicitly declare the variable with automatic storage. It is the default storage class for the variables declared in a block.
Eg: auto int x;

Register: When a variable is declared using register as it storage class, it is stored in CPU register instead of RAM. Since the variable in RAM, the maximum size of the variable is equal to the register size. It is used when a programmer need quick access to the variable.
Eg: Register int x;

External: It is used to give a reference of global variable that is visible to all the program files.
Eg: Extern int x;

Static: While auto is the default storage class for all local variables, static is the default storage class for global variables. Static variable have a life over the entire program. ie., memory for the static variables is allocated when the program begins running and is freed when the program terminates.
Eg: Static int x=10;

5. **Define function. Why they are needed? What are its types?**

A function is a group of statements that together perform a task. C enables its programmer to break-up a program into segment commonly known as function, each of which can be written more or less independently of the others. Every function in the program supposed to perform a well defined task. It is needed because, dividing the program into separate well defined functions facilitates each function to be written and tested separately. Understanding, coding and testing multiple separate function are easier than doing the same for one huge function.

Types:

1. Built in Functions
2. User Defined functions

6. **Write short notes on pointers, null pointers, and generic pointers. How to declare and initialize these pointers?**

Pointer:

Pointer is a variable which holds the address of another variable.

Pointer Declaration:

datatype *variable-name;

Example:
int *x, c=5;
x=&a;
Null Pointer: It is a special pointer value that is known not to point anywhere, this menas that the null pointer does not to point to any valid memory address.
Eg: int * ptr=NULL;
Generic pointer: It is a pointer variable that has void as its data type. The void pointer, or the generic pointer, is a special tyoe of pointer that can be pointed at variable of any data type.
For example, void * ptr;

7. **What is an Array? What are its types? List out the operations performed on an array? Write down the general form of calculating length of the Array.**

Array:

An array is a derived data type, which is a collection of element of same type stored in a physically continues memory location. Example, int marks [10];

Types of arrays:

1) Single dimensional array. eg: int a[5];
2) Multi dimensional array. eg: int a[5] [5];

List of Operations performed on array:

- Insertion: insert an element into the array.
- Deletion: Delete an element from the array.
- Traversal: Accesses each element of the array.
- Sort: Re-arranges the array element in a specific order.
- Search: Searches the key value in the array.

Calculating the length of the Array:

The length of the array is given by the number of elements stored in it. The general formula to calculate the length the array is, length = index of the last element – index of the first element +1.

8. **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.

9. **Distinguish between #ifdef and #if directives.**

#ifdef

#ifdef is the simplest sort of conditional preprocessor directive and is used to check for the existence of macro definitions. Its syntax can be given as:

#ifdef *MACRO*

controlled text

#endif

The #if Directive

The #if directive is used to control the compilation of portions of a source file. If the specified condition (after the #if) has a nonzero value, the controlled text immediately following the #if directive is retained in the translation unit. The #if directive in its simplest form consists of

#if condition

controlled text

#endif

While using #if directive, make sure that each #if directive must be matched by a closing #endif directive. Any number of #elif directives can appear between the #if and #endif directives, but at most one #else directive is allowed. However, the #else directive (if present) must be the last directive before #endif.

10. **Define the terms: File inclusion, Conditional and Pragma Directives.**

File inclusion Directive:

When the preprocessor finds an #include directive it replaces it by the entire content of the specified file. There are two ways to specify a file to be included:

#include "file"

#include <file>

Conditional directive:

A conditional directive is used instruct the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. The preprocessor conditional directives can test arithmetic expressions, or whether a name is defined as a macro, etc. The if-else directives #if, #ifdef, #ifndef, #else, #elif and #endif can be used for conditional compilation.

Pragma Directive:

The #pragma directive is a compiler-specific directive, used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status. The syntax of using a pragma directive can be given as: #pragma *string.*

The commonly used forms of the pragma directives are:

1. #pragma option (-C,-C-,-G,-G-,-r,-r-,-a,-a-)
2. #pragma warn (dup,voi,rvl,par,pia,rch,aus)
3. #pragma startup and #pragma exit

16 MARKS

1. **(i)Explain about passing parameters to the function with and without using pointers with suitable example.(or) Explain about Call by Value and Call by Reference with an example program.**
   (ii)What is Recursion? Explain Different types of recursion.

   (i)PASSING PARAMETERS TO THE FUNCTION

   More than one value can be indirectly returned to the calling function by making the use of pointers. The pointers can also be used to pass arguments to a function. There are two ways in which arguments or parameters can be passed to the called function.

- Call by value (pass by value) in which values of the variables are passed by the calling function to the called function.
- Call by reference (pass by reference) in which address of the variables are passed by the calling function to the called function.

Call by value

- In the Call by Value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.
- If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

Example:

```
#include<stdio.h>
void add( int n);
int main()
{
        int num = 2;
        printf("\n The value of num before calling the function = %d", num);
        add(num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add(int n)
{
        n = n + 10;
        printf("\n The value of num in the called function = %d", n);
}
```

The output of this program is:

```
The value of num before calling the function = 2
```

The value of num in the called function = 20
The value of num after calling the function = 2

CALL BY REFERENCE

- When the calling function passes arguments to the called function using call by value method, the only way to return the modified value of the argument to the caller is explicitly using the return statement. The better option when a function can modify the value of the argument is to pass arguments using call by reference technique.
- In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it received are visible by the calling program.
- To indicate that an argument is passed using call by reference, an ampersand sign (&) is placed after the type in the parameter list. This way, changes made to that parameter in the called function body will then be reflected in its value in the calling program.

Example:

```
#include<stdio.h>
void add( int *);
int main()
{
        int num = 2;
        printf("\n The value of num before calling the function = %d", num);
        add(&num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add( int *n)
{
        *n =* n + 10;
        printf("\n The value of num in the called function = %d", *n);
}
```

The output of this program is:

The value of num before calling the function = 2
The value of num in the called function = 20
The value of num after calling the function = 20

(ii)What is Recursion? Explain Different types of recursion.

- A recursive function is a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases, they are

  base case, in which the problem is simple enough to be solved directly without making any further calls to the same function

  recursive case, in which first the problem at hand is divided into simpler sub parts. Second the function calls itself but with sub parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of a smaller and more easily solvable problem. In recursive function, complicated problem is defined in terms of simpler problems and the simplest problem is given explicitly.

TYPES OF RECURSION

Any recursive function can be characterized based on:

1. whether the function calls itself directly or indirectly (direct or indirect recursion).
2. whether any operation is pending at each recursive call (tail-recursive or not).
3. the structure of the calling pattern (linear or tree-recursive).

DIRECT RECURSION

A function is said to be *directly* recursive if it explicitly calls itself. For example, consider the function given below.

```
int Func( int n)
{
        if(n==0)
                retrun n;
        return (Func(n-1));
}
```

INDIRECT RECURSION

A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
                return n;
        return Func2(n);
}
```

```
int Func2(int x)
{
        return Func1(x-1);
}
```

TAIL RECURSION

- A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

```
int Fact(n)
{
 return Fact1(n, 1);
}
```

```
int Fact1(int n, int res)
{
        if (n==1)
           return res;
        return Fact1(n-1, n*res);
}
```

LINEAR AND TREE RECURSION

- Recursive functions can also be characterized depending on the way in which the recursion grows- in a linear fashion or forming a tree structure.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function. For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to Fact.

- On the contrary, a recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.

2. **What is a pointer? Explain about (i) pointer to functions (Function pointer) (ii)calling a function using function pointer ,(iii)passing a Function Pointer as an Arguments to a Function and (iv) Array of pointers (v) Pointer to a Pointers (vi) Pointer to an Array with an example.**

POINTER

Pointer is a variable which holds the address of another variable.

Pointer Declaration:

datatype *variable-name;

Example:

int *x, c=5;
x=&a;

Uses of Pointers

Pointers are used to return more than one value to the function
· Pointers are more efficient in handling the data in arrays
· Pointers reduce the length and complexity of the program
· They increase the execution speed
· The pointers save data storage space in memory

(i) POINTER TO FUNCTION (Function pointer)

- This is a useful technique for passing a function as an argument to another function.
- In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name.

```
/* pointer to function returning  int */
int (*func)(int a, float b);
```

- If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by using its name.

(ii) CALLING A FUNCTION USING FUNCTION POINTER

A function pointer can be used to call a function in any of the following two ways:

1. By explicitly dereferencing it using the dereference operator (*).
2. By using its name instead of the function's name

When a pointer to a function is declared, it can be **called** using one of two forms:

(*func)(1,2); OR func(1,2);

Example:

```
#include <stdio.h>
void print(int n);
main()
{
        void (*fp)(int);
        fp = print;
        (*fp)(10);
        fp(20);
        return 0;
}
void print(int value)
{
        printf("\n %d", value);
}
```

(iii) **PASSING A FUNCTION POINTER AS AN ARGUMENT TO A FUNCTION**

A function pointer can be passed as a function's calling argument. This is done when you want to pass a pointer to a callback function.

Example:

```
include <stdio.h>
int add(int, int);
int subt(int, int);
int operate(int (*operate_fp) (int, int), int, int);
main()
{       int result;
        result = operate(add, 9, 7);
        printf ("\n Addition Result = %d", result);
        result = operate(sub, 9, 7);
        printf ("\n Subtraction Result = %d", result);
}
int add (int a, int b)
```

```
{
 return (a+b);
}
int subtract (int a, int b)
{
  return (a-b);
}
int operate(int (*operate_fp) (int, int), int a, int b)
{
        int result;
        result = (*operate_fp) (a,b);
        return result;
}
```

(iv) ARRAY OF POINTERS

An array of pointers is a collection of addresses. The addresses in an array of pointers could be the addresses of isolated variables or the addresses of array elements or any other addresses. The only constraint is that all the pointers in an array must be of the same type.

Example:

```
#include<stdio.h>

main()

{

int a=10,b=20,c=30;

int *arr[3]={&a,&b,&c};

printf("the values of variables are:\n");

printf("%d%d%d\n",a,b,c);

printf("%d%d%d\n",*arr[0],*arr[1],*arr[2]);

}
```

(v) **POINTERS TO A POINTERS**

- You can use pointers that point to pointers. The pointers in turn, point to data (or even to other pointers). To declare pointers to pointers just add an asterisk (*) for each level of reference.

  For example, if we have:

  ```
  int x=10;
  ```

```
int  *px, **ppx;
px=&x;
ppx=&px;

Now if we write,
printf("\n %d", **ppx);
Then it would print 10, the value of x.
```

(vi) POINTER TO AN ARRAY

It is possible to create a pointer that points to a complete array instead of pointing to the individual elements of an array or isolated variables. Such pointer is known as pointer to an array.

Example:

```
#include<stdio.h>

main()

{

int arr[2][2]={{2,1},{3,5}};

int (*ptr)[2]=arr;

printf("%d\n",arr[0][0]);

printf("%p\n",arr[0]);

printf("%p\n",ptr+1);

}
```

Output:

```
2

234F:2234

234F:2238
```

3. Explain briefly about Various Control Structures in C with suitable examples.

IF STATEMENTS

The if statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```
if( expression )
program statement;
```

Example:

```
if( students < 65 )
++student_count;
```

IF ELSE

The general format for these is,

```
if( condition 1 )
statement1;
else if( condition 2 )
 statement2;
else if( condition 3 )
 statement3;
else
statement4;
```

The else clause allows action to be taken where the condition evaluates as false (zero).

The following program uses an if else statement to validate the users input to be in the range 1-10.

Example:

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 )
{
printf("Enter a number between 1 and 10 -->");
scanf("%d",&number);
if( number < 1 )
{
printf("Number is below 1. Please re-enter\n");
valid = 0;
}
else if( number > 10 )
{
printf("Number is above 10. Please re-enter\n");
valid = 0;
}
```

```
else
valid = 1;
}
printf("The number is %d\n", number );
}
```

NESTED IF ELSE

Example:

```
#include <stdio.h>
main()
{
int invalid_operator = 0;
char operator;
float number1, number2, result;
printf("Enter two numbers and an operator in the format\n");
printf(" number1 operator number2\n");
scanf("%f %c %f", &number1, &operator, &number2);
if(operator == '*')
result = number1 * number2;
else if(operator == '/')
result =number1 / number2;
else if(operator == '+')
result = number1 + number2;
else if(operator == '-')
result = number1 - number2;
else
invalid_operator = 1;
if( invalid_operator != 1 )
printf("%f %c %f is %f\n", number1, operator, number2, result );
else
printf("Invalid operator.\n");
```

SWITCH CASE:

The switch case statement is a better way of writing a program when a series of if elses occurs. The general format for this is,

```
switch ( expression )
{
case value 1:
program statement;
program statement;
.....
```

```
        break;
        case value n:
        program statement;
        .......
        break;
        default:
        .......
        .......
        break;
        }
```

The keyword break must be included at the end of each case statement. The default clause is optional, and is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Example:

```
#include <stdio.h>
main()
{
int menu, numb1, numb2, total;
printf("enter in two numbers -->");
scanf("%d %d", &numb1, &numb2 );
 printf("enter in choice\n");
printf("1=addition\n");
printf("2=subtraction\n");
scanf("%d", &menu );
switch( menu )
{
case 1:
total = numb1 + numb2;
break;
case 2:
total = numb1 -numb2;
break;
default:
printf("Invalid option selected\n");
}
if( menu == 1 )
printf("%d plus %d is %d\n", numb1, numb2, total );
else if( menu == 2 )
printf("%d minus %d is %d\n", numb1, numb2, total );
}
```

The above program uses a switch statement to validate and select upon the users input choice, simulating a simple menu of choices.

FOR LOOPS

The basic format of the for statement is,

```
        for( start condition; continue condition; re-evaulation )
        program statement;
```

Example:

```
/* sample program using a for statement*/
#include <stdio.h>
main()
{
int count;
for( count = 1; count <= 10; count = count + 1 )
printf("%d ", count );
printf("\n");
}
```

The program declares an integer variable count. The first part of the for statement count = 1;initialises the value of count to 1. The for loop continues whilst the condition
count <= 10; evaluates as TRUE. As the variable count has just been initialised to 1, this condition is TRUE and so the program statement printf("%d ", count); is executed, which prints the value of count to the screen, followed by a space character. Next, the remaining statement of the for is executed count = count + 1 , which adds one to the current value of count. Control now passes back to the conditional test, count <= 10; which evaluates as true, so the program statement printf("%d ", count); is executed. Count is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test count <= 10; evaluates as FALSE, and the for loop terminates, and program control passes to the statement printf("\n"); which prints a newline, and then the program terminates, as there are no more statements left to execute.

THE WHILE STATEMENT

The while provides a mechanism for repeating C statements whilst a condition is true. Its format is,

```
while( condition )
program statement;
```

Somewhere within the body of the while loop a statement must alter the value of the condition to allow the loop to finish.

Example:

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

The above program uses a while loop to repeat the statements
printf("%d\n", loop);
++loop;
while the value of the variable loop is less than or equal to 10.

Note how the variable upon which the while is dependant is initialised prior to the while statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the while loop will terminate. This program is functionally equivalent to the earlier for program which counted to ten.

THE DO WHILE STATEMENT

The do { } while statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero).
The loop is executed as least once.

Example:

```
#include<stdio.h>
main()
{
int value, r_digit;
printf("Enter the number to be reversed.\n");
scanf("%d", &value);
do
{
r_digit = value % 10;
printf("%d", r_digit); value = value / 10;
}
while( value !=0 );
printf("\n");
}
```

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digits into the variable r_digit. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

4. (i) Write a program to calculate the sum of numbers from m to n.

```
#include<stdio.h>
int main()
{
int n,m,sum=0;
clrscr();
printf("\n enter the value of m:");
scanf("%d",&m);
printf("\n enter the value of n:");
scanf("%d",&n);
```

```
while(m<=n)
{
    sum=sum+m;
    m=m+1;
}
printf("\n the sum of numbers from %d to %d=%d", m ,n ,sum);
return 0;
}
```

5)i) Write a program to find the greatest amongst three numbers using conditional operator.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int num1,num2,num3,large;
clrscr();
printf("\n enter the first number:");
scanf("%d",&num1);
printf("\n enter the second number:");
scanf("%d",&num2);
printf("\n enter the third number:");
scanf("%d",&num3);
large=num1>num2?(num1>num3?num1:num3):(num2>num3?num2:num3);
printf("\n the largest number is:%d",large);
return 0;
}
```

(ii) Write a program to determine whether an entered character is vowel or not, using switch case.

```
#include<stdio.h>
#include<conio.h>
int main()
{
char ch;
printf("\n enter any character:");
switch(ch)
{
case 'A':
case 'a':
        printf("\n %c is vowel",ch);
        break;
case 'E':
case 'e':
        printf("\n %c is vowel",ch);
        break;
```

```
case ‘I’:
case ‘i’:
        printf(“\n %c is vowel”,ch);
        break;
case ‘O’:
case ‘o’:
        printf(“\n %c is vowel”,ch);
        break;
case ‘U’:
case ‘u’:
        printf(“\n %c is vowel”,ch);
        break;
default:
        printf(“\n %c is not a vowel”,ch);
}
return 0;
}
```

(iv) Write a program to print the Fibonacci series using recursion.

```
#include<stdio.h>

#include<conio.h>

int fibonacci(int);

main()

        {
        int n;

        printf(“\n enter the number of terms in the series:”);

        scanf(“%d”,&n);

        for(i=0;i<n;i++)

                printf(“\n fibonacci (%d)=%d”,i,fibonacci(i));

        return 0;

}

int fibonacci(int num)

{

        if(num<=2)

                return 1;
```

```
        else
                return(fibonacci(num-1)+fibonacci(num-2));
    }
```

6.Write a program to insert a number at a given location in an array.

```
#include<stdio.h>
#include<conio.h>
main()
{
int i,n,num,pos,arr[10];
clrscr();
printf("\n enter the number of elements in the array:");
scanf("%d",&arr[i]);
for(i=0;i<n;i++)
{
printf("\n arr[%d]=:",i)
scanf("%d",&arr[i]);
}
printf("\n enter the number to be inserted:");
scanf(""%d",&num);
printf("enter the position at which the number has to be added:");
scanf("%d",&pos);
for(i=n;i>=pos;i--)
{
arr[i+1]=arr[i];
arr[pos]=num;
printf("\n the array after insertion of %d is:",num);
```

```
for(i=0;i<n+1;i++)
{
printf("\n arr[%d]=%d",i,arr[i]);
getch();}
```

(ii)Write a Program to merge two unsorted Arrays.

Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence,the merged array contains the contents of the first array followed by the contents of the second array.

```
#include<stdio.h>
#include<conio.h>
main()
{
int arr1[10],arr2[10],arr3[20];
int i,n1,n2,m,index=0;
clrscr();
printf("\n printf("\n enter the number of elements in the array1:");
scanf("%d",&n1);
printf("\n enter the elements of the array1:");
for(i=0;i<n1;i++)
{
printf("\n arr1[%d]=",i);
scanf("%d",arr1[i]);
}
printf("\n enter the number of elements in the array2:");
scanf("%d",&n2);
printf("\n enter the elements of the array2:");
```

```
for(i=0;i<n2;i++)
{
printf("\n arr2[%d]=",i);
scanf("%d",arr2[i]);
}
m=n1+n2;
for(i=0;i<n1;i++)
{
arr3[index]=arr1[i];
index++;
}
for(i=0;i<n2;i++)
{
arr3[index]=arr2[i];
index++;
}
printf("\n\n the merged array is");
for(i=0;i<m;i++)
printf("\n arr[%d]=%d",i,arr3[i]);
getch();
}
```

7. Explain in detail about various Pre-processor directives with suitable examples.

- The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of preprocessor directive which is placed in the source program before the main().
- Before the source code is passed through the compiler, it is examined by the preprocessor for any preprocessor directives. In case, the program has some

preprocessor directives, appropriate actions are taken (and the source program is handed over to the compiler.

- The preprocessor directives are always preceded by a hash sign (#).
- The preprocessor is executed before the actual compilation of program code begins. Therefore, the preprocessor expands all the directives and take the corresponding actions before any code is generated by the program statements.
- No semicolon (;) can be placed at the end of a preprocessor directive.

The advantages of using preprocessor directives in a C program include:

- Program becomes readable and easy to understand
- Program can be easily modified or updated
- Program becomes portable as preprocessor directives makes it easy to compile the program in different execution environments
- Due to the aforesaid reason the program also becomes more efficient to use.

TYPES OF PREPROCESSOR DIRECTIVES

1. Macro replacement directive (#define,#undef)
2. Source file inclusion directive(#include)
3. Line directive(#line)
4. Error directive(#error)
5. Pragma directive(#pragma)
6. Conditional compilation directives(#if,#else,#elif,#endif,#ifdef,#ifndef)

1. MACRO REPLACEMENT DIRECTIVE (#define,#undef)

A Macro is a facility provided by the C preprocessor, by which a token can be replaced by the User-defined sequence of characters. Macro name are generally written in upper case.

#undef

As the name suggests, the #undef directive undefines or removes a macro name previously created with #define. Undefining a macro means to cancel its definition. This is done by writing #undef followed by the macro name that has to be undefined.

#define

- To define preprocessor macros we use #define. The #define statement is also known as macro definition or simply a macro. There are two types of macros- object like macro and function like macro.

Types of Macro:

1. Macro without Arguments, also called as Object-like macros.

2. Macro with Arguments, also called as Function-like macros.

Object like macro

- An *object-like macro* is a simple identifier which will be replaced by a code fragment. They are usually used to give symbolic names to numeric constants. Object like macros do not take any argument. It is the same what we have been using to declare constants using #define directive. The general syntax of defining a macro can be given as:

  #define identifier string
- The preprocessor replaces every occurrence of the identifier in the source code by a string.

Example

```
#include<stdio.h>
 #define PI 3.14
main()
{
int rad=5;
Printf("area of circle is %f",PI*rad*rad);
}
```

Function-like macros

- They are used to stimulate functions.
- When a function is stimulated using a macro, the macro definition replaces the function definition.
- The name of the macro serves as the header and the macro body serves as the function body. The name of the macro will then be used to replace the function call.
- **The function-like macro includes a list of parameters.**
- References to such macros look like function calls. However, when a macro is referenced, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments, and the text is inserted into the program stream. Therefore, macros are considered to be much more efficient than functions as they avoid the overhead involved in calling a function.
- The **syntax** of defining a function like macro can be given as

  # define identifier(arg1,arg2,...argn) string

Example

```
#include<stdio.h>
#include<conio.h>
#define s(x) x*x
Void main()
{
Clrscr();
Printf("%d",s(5));
}
```

Output:

25

2. SOURCE FILE INCLUSION DIRECTIVE

#include

- An external file containing function, variables or macro definitions can be included as a part of our program. This avoids the effort to re-write the code that is already written.
- The #include directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.
- The #include directive can be used in two forms. Both forms makes the preprocessor insert the entire contents of the specified file into the source code of our program. However, the difference between the two is the way in which they search for the specified.
  #include <*filename*>
- This variant is used for system header files. When we include a file using angular brackets, a search is made for the file named *filename* in a standard list of system directories.
  #include "*filename*"
- This variant is used for header files of your own program. When we include a file using double quotes, the preprocessor searches the file named *filename* first in the directory containing the current file, then in the quote directories and then the same directories used for <*filename*>.

Example:

File name 1: T.cpp

```
#include<stdio.h>
#include<conio.h>
#include “max.cpp”
void main()
{
clrscr();
printf(“%d”,max());
}
```

File name 2: max.cpp

```
Void max()
{
Int a=5,b=3,m;
m=(a>b)?a:b;
Printf(“%d”,m);
}
```

5. LINE DIRECTIVE

The line directive is used to reset the line number. The line directive is used for the purpose of error diagnostics. It has two forms:

1. #line constant

2. #line constant “filename”

#line

- Compile the following C program

```
#include<stdio.h>
main()
{       int a=10:
        printf("%d", a);
}
```

- The above program has a compile time error because instead of a semi-colon there is a colon that ends line int a = 10. So when you compile this program an error is generated during the compiling process and the compiler will show an error message with references to the name of the file where the error happened and a line number. This makes it easy to detect the erroneous code and rectify it.
- The #line directive enables the users to control the line numbers within the code files as well as the file name that we want that appears when an error takes place. The syntax of #line directive is:
- #line line_number filename
- Here, line_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.
- Filename is an optional parameter that redefines the file name that will appear in case an error occurs. The filename must be enclosed within double quotes. If no filename is specified then the compiler will show the original file name. For example:

```
#include<stdio.h>
main()
{       #line 10 "Error.C"
        int a=10:
        #line 20
        printf("%d, a);
}
```

6. PRAGMA DIRECTIVES

The #pragma directive is a compiler-specific directive, used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status. A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing.

- The #pragma directive is used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole.
- The effect of pragma will be applied from the point where it is included to the end of the compilation unit or until another pragma changes its status.

- A #pragma directive is an instruction to the compiler and is usually ignored during preprocessing.
- The syntax of using a pragma directive can be given as:

  #pragma *string*

  The commonly used forms of the pragma directives are:

  1. #pragma option (-C,-C-,-G,-G-,-r,-r-,-a,-a-)
  2. #pragma warn (dup,voi,rvl,par,pia,rch,aus)
  3. #pragma startup and #pragma exit

  Example:

```
#include<stdio.h>
#include<conio.h>
#pragma warn –rvl
main()
{
Printf("hi");
}
```

  Output: **hi**

  Note: Without the pragma directive, the output is, **Function should return a value.**

7. CONDITIONAL DIRECTIVES

- A conditional directive is used instruct the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. The preprocessor conditional directives can test arithmetic expressions, or whether a name is defined as a macro, etc.
- Conditional preprocessor directives can be used in the following situations:
- A program may need to use different code depending on the machine or operating system it is to run on.
- The conditional preprocessor directive is very useful when you want to compile the same source file into two different programs. While one program might make frequent time-consuming consistency checks on its intermediate data, or print the values of those data for debugging, the other program, on the other hand can avoid such checks.
- The conditional preprocessor directives can be used to exclude code from the program whose condition is always false but is needed to keep it as a sort of comment for future reference.

#ifdef

- #ifdef is the simplest sort of conditional preprocessor directive and is used to check for the existence of macro definitions.
- Its syntax can be given as:

```
#ifdef MACRO
        controlled text
#endif
#ifdef MAX
int STACK[MAX];
```

#endif

#ifndef

The #ifndef directive is the opposite of #ifdef directive. It checks whether the MACRO has not been defined or if its definition has been removed with #undef.

#ifndef is successful and returns a non-zero value if the MACRO has not been defined. Otherwise in case of failure, that is when the MACRO has already been defined, #ifndef returns false (0).

The general format to use #ifndef is the same as for #ifdef:

#ifndef *MACRO*
controlled text
#endif

The #if Directive

- The #if directive is used to control the compilation of portions of a source file. If the specified condition (after the #if) has a nonzero value, the controlled text immediately following the #if directive is retained in the translation unit.
- The #if directive in its simplest form consists of

#if condition
controlled text
#endif

- While using #if directive, make sure that each #if directive must be matched by a closing #endif directive. Any number of #elif directives can appear between the #if and #endif directives, but at most one #else directive is allowed. However, the #else directive (if present) must be the last directive before #endif.

The #else Directive

- The #else directive can be used within the controlled text of an #if directive to provide alternative text to be used if the condition is false.
- The general format of #else directive can be given as:

```
#if condition
        Controlled text1
#else
        Controlled text2
#endif
```

The #elif Directive

- The #elif directive is used when there are more than two possible alternatives. The #elif directive like the #else directive is embedded within the #if directive and has the following syntax:

```
#if condition
        Controlled text1
#elif new_condition
        Controlled text2
#else
        Controlled text3
#endif
```

The #endif Directive

- The general syntax of #endif preprocessor directive which is used to end the conditional compilation directive can be given as:

#endif

- The #endif directive ends the scope of the #if , #ifdef , #ifndef , #else , or #elif directives.

8. #error DIRECTIVE

- The #error directive is used to produce compiler-time error messages. The syntax of this directive is:
  #error string
- The error messages include the argument *string.* The #error directive is usually used to detect programmer inconsistencies and violation of constraints during preprocessing.
- When #error directive is encountered, the compilation process terminates and the message specified in string is printed to stderr.

Example:

```
#ifndef SQUARE
#error MACRO not defined.
#endif
#ifndef VERSION
#error Version number is not specified.
#endifsss
```

UNIT –II

PART A

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 name represents the address of the starting element.	A structrure name is known as tag. It is a Shorthand notation of the declaration.
An array cannot have bit fields.	A structure may contain bit fields.

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. 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;
```

4. 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.
5. What are the storage classes available in C?
There are following storage classes which can be used in a C Program
Auto,register,static,extern
6.Write a note on File.
A file is a collection of related information normally representing programs, both source and object forms and data. Data may be numeric, alphabetic or alphanumeric. A file can also be defined as a sequence of bits, bytes, lines or records whose meaning is defined by the programmer. Operations such as create, open, read, write and close are performed on files.
A data field is an elementary unit that stores a single fact.
A record is a collection of related data fields that is sen as a single unit from the application
A File is a collection of related records.
A Key the part of data BY WHICH IT IS STORED,INDEXED,CROSS REFERENCE,ETC.
A Indax file that stores keys and index into another file.

7.what are Types of File Handling in C?
The file handling in c can be categorized in two types-

1. **High level (Standard files or stream oriented files)-** High level file handling is managed by library function. High level file handling commonly used because it is easier and hide most of the details from the programmer.
2. **Low level (system oriented files)-** low level files handling is managed by system calls.

8.What are the Step for file operation in C programming?

1. Open a file
2. Read the file or write data in the file
3. Close the file

9.Define File modes?

File mode	Description
R	Open a text file for reading.
W	Create a text file for writing, if it exists, it is overwritten.
A	Open a text file and append text to the end of the file.
Rb	Open a binary file for reading.
Wb	Create a binary file for writing, if it exists, it is overwritten.
Ab	Open a binary file and append data to the end of the file

10.Define End of file?

The file reading function need to know the end of file so that they can stop reading .when the end of file is reached , the operating signal send s an end of file signal to the program. when the program receive this signal , the file reading function return EOF which is constant defined in the file stdio.h an its value -1 . EOF is an integer value so make sure the return value of the function Ithe ares assigned to a integer variables.

PART B

1.Explain in detailed about formatted I/O?

Formatted Output

The printf functions provide formatted output conversion.

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

fprintf converts and writes output to stream under the control of format. The return value is the number of characters written, or negative if an error occurred.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to fprintf. Each conversion specification begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, in order:

· Flags (in any order), which modify the specification:

o -, which specifies left adjustment of the converted argument in its field.

o +, which specifies that the number will always be printed with a sign.

o *space*: if the first character is not a sign, a space will be prefixed.

o 0: for numeric conversions, specifies padding to the field width with leading zeros.

o #, which specifies an alternate output form. For o, the first digit will become zero. For x or X, 0x or 0X will be prefixed to a non-zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed.

· A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is 0 if the zero padding flag is present.

· A period, which separates the field width from the precision.

· A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for

e, E, or f conversions, or the number of significant digits for g or G conversion, or the number of digits to be printed for an integer (leading 0s will be added to make up the necessary width).

· A length modifier h, l (letter ell), or L. ``h'' indicates that the corresponding argument is to be printed as a short or unsigned short; ``l'' indicates that the argument is a long or unsigned long, ``L'' indicates that the argument is a long double.

Width or precision or both may be specified as *, in which case the value is computed by converting the next argument(s), which must be int.

The conversion characters and their meanings are shown in Table B.1. If the character after the % is not a conversion character, the behavior is undefined.

Character Argument type; Printed As

d,i int; signed decimal notation.

o int; unsigned octal notation (without a leading zero).

x,X

unsigned int; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.

u int; unsigned decimal notation.

c int; single character, after conversion to unsigned char

s

char *; characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.

f

double; decimal notation of the form [-]*mmm.ddd*, where the number of *d*'s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.

e,E

double; decimal notation of the form [-]*m.dddddd*e+/-*xx* or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.

g,G

double; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal point are not printed.

p void *; print as a pointer (implementation-dependent representation).

n

int *; the number of characters written so far by this call to printf is *written into* the argument. No argument is converted.

% no argument is converted; print a %

int printf(const char *format, ...)

printf(...) is equivalent to fprintf(stdout, ...).

int sprintf(char *s, const char *format, ...)

sprintf is the same as printf except that the output is written into the string s, terminated with '\0'. s must be big enough to hold the result. The return count does not include the '\0'.

int vprintf(const char *format, va_list arg)

int vfprintf(FILE *stream, const char *format, va_list arg)

int vsprintf(char *s, const char *format, va_list arg)

The functions vprintf, vfprintf, and vsprintf are equivalent to the corresponding printf functions, except that the variable argument list is replaced by arg, which has been initialized by the va_start macro and perhaps va_arg calls. See the discussion of <stdarg.h>

Formatted Input

The scanf function deals with formatted input conversion.

int fscanf(FILE *stream, const char *format, ...)

fscanf reads from stream under control of format, and assigns converted values through subsequent arguments, *each of which must be a pointer*. It returns when format is exhausted. fscanf returns EOF if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

· Blanks or tabs, which are not ignored.

· Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.

· Conversion specifications, consisting of a %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by *, as in %*s, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that scanf will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B.2.

The conversion characters d, i, n, o, u, and x may be preceded by h if the argument is a pointer to short rather than int, or by l (letter ell) if the argument is a pointer to long. The conversion characters e, f, and g may be preceded by l if a pointer to double rather than float is in the argument list, and by L if a pointer to a long double.

Character Input Data; Argument type

d decimal integer; int*

i

integer; int*. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).

o octal integer (with or without leading zero); int *.

u unsigned decimal integer; unsigned int *.

x hexadecimal integer (with or without leading 0x or 0X); int*.

c

characters; char*. The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No '\0' is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use %1s.

sstring of non-white space characters (not quot ed); char *, pointing to an array of characters large enough to hold the string and a terminating '\0' that will be added.

e,f,g

floating-point number; float *. The input format for float's is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an E or e followed by a possibly signed integer.

p pointer value as printed by printf("%p");, void *.

n

writes into the argument the number of characters read so far by this call; int *.
No input is read. The converted item count is not incremented.
[...]
matches the longest non-empty string of input characters from the set between
brackets; char *. A '\0' is added. []...] includes] in the set.
225
[^...]
matches the longest non-empty string of in put characters *not* from the set between
brackets; char *. A '\0' is added. [^]...] includes] in the set.
% literal %; no assignment is made.
int scanf(const char *format, ...)
scanf(...) is identical to fscanf(stdin, ...).
int sscanf(const char *s, const char *format, ...)
sscanf(s, ...) is equivalent to scanf(...) except that the input characters are
taken from the string s.

2.Explain the structure and Union Declaration?

A structure is an object consisting of a sequence of named members of various types. A union is an object that contains, at different times, any of several members of various types.
Structure and union specifiers have the same form.
struct-or-union-specifier:
struct-or-union identifieropt{ *struct-declaration-list* }
struct-or-union identifier
struct-or-union:
struct
union
A struct-declaration-list is a sequence of declarations for the members of the structure or union:
struct-declaration-list:
struct declaration
struct-declaration-list struct declaration
struct-declaration: *specifier-qualifier-list struct-declarator-list*;
specifier-qualifier-list:
type-specifier specifier -qualifier-listopt
type-qualifier specifier -qualifier-listopt
struct-declarator -list:
struct-declarator
struct-declarator-list , *struct-declarator*
Usually, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *bit-field*; its length is set off from the declarator for the field name by a colon.
struct-declarator :
declarator declarator opt : *constant-expression*
A type specifier of the form
struct-or-union identifier { *struct-declaration-list* }
declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same or an inner scope may refer to the sa specifier without the list:
struct-or-union identifier
If a specifier with a tag but without a list appears when the tag is not declared, an *incomplete type* is specified. Objects with an incomplete structure or union type may be mentioned in

contexts where their size is not needed, for example in declarations (not definitions), for specifying a pointer, or for creating a typedef, but not otherwise. The type becomes complete on occurrence of a subsequent specifier with that tag, and containing a declaration list. Even list, and becomes complete only at the } terminating the specifier.

A structure may not contain a member of incomplete type. Therefore, it is impossible to declare a structure or union containing an instance of itself. However, besides giving a name to the structure or union type, tags allow definition of self-referential structures; a structure or union may contain a pointer to an instance of itself, because pointers to incomplete types may be declared.

A very special rule applies to declarations of the form

struct-or-union identifier;

that declare a structure or union, but have no declaration list and no declarators.

declaration makes the identifier the tag of a new, incompletely-typed structure or union in the current scope.

This recondite is new with ANSI. It is intended to deal with mutually -recursive structures declared in an

inner scope, but whose tags might already be declared in the outer scope.

A structure or union specifier with a list but no tag creates a unique type; it can be referred to directly only in the declaration of which it is a part.

The names of members and tags do not conflict with each other or with ordinary variables. A member name may not appear twice in the same structure or union, but the same member name may be used in different structures or unions.

In the first edition of this book, the names of structure and union members were not associated with their parent. However, this association became common in compilers well before the ANSI standard.

A non-field member of a structure or union may have any object type. A field member (which need not have a declarator and thus may be unnamed) has type int, unsigned int, or signed int, and is interpreted as an object of integral type of the specified length in bits; whether an int field is treated as signed is implementation-dependent. Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction. When a field following another field will not fit into a partially-filled storage unit, it may be split between units, or the unit may be padded. An unnamed field with width 0 forces this padding, so that the next field will begin at the edge of the next allocation unit.

The ANSI standard makes fields even more implementation-dependent than did the first edition. It is advisable to read the language rules for storing bit-fields as ``implementation -dependent'' without qualification. Structures with bit-fields may be used as a portable way of attempting to reduce the storage required for a structure (with the probable cost of increasing the instruction space, and time,

needed to access the fields), or as a non-portable way to describe a storage layout known at the bitlevel.

In the second case, it is necessary to understand the rules of the local implementation.

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member.

A simple example of a structure declaration is

```
struct tnode {
char tword[20];
int count;
struct tnode *left;
struct tnode *right;
```

}
which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has bene given, the declaration
struct tnode s, *sp;
declares s to be a structure of the given sort, and sp to be a pointer to a structure of the given sort. With these declarations, the expression
sp->count
refers to the count field of the structure to which sp points;
s.left
refers to the left subtree pointer of the structure s, and
s.right->tword[0]
refers to the first character of the tword member of the right subtree of s.
In general, a member of a union may not be inspected unless the value of the union has been assigned using the same member. However, one special guarantee simplifies the use of unions: if a union contains several structures that share a common initial sequence, and the union currently contains one of these structures, it is permitted to refer to the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union {
struct {
int type;
} n;
struct {
int type;
int intnode;
} ni;
struct {
int type;
float floatnode;
} nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
... sin(u.nf.floatnode) ...
```

3..Write a program to copy content of one file to another using command line arguments

```
int main(int argc, char *argv[])
{
FILE * ifil,*ofil;
char buffer [100];
if (argc!=3){
printf("Usage: copyfile inFile outFile");
return 0;
}
ifil = fopen (argv[1], "r");
ofil = fopen (argv[2], "w");
if (ifil == NULL) perror ("Error opening input file");
else
{
```

```
while ( ! feof (ifil) )
{
fgets (buffer , 100 , ifil);
fputs (buffer , ofil);
}
fclose (ifil);
fclose (ofil);
}
return 0;
}
```

4..Write a program to write text in a file. Read the text from the file from end of file. Display the contents of file in reverse order.

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

void read_file(FILE *fileptr)
{
   char currentchar = '\0';
   int size = 0;

   while( currentchar != '\n' )
   {
      currentchar = fgetc(fileptr); printf("%c\n", currentchar);
      fseek(fileptr, -2, SEEK_CUR);
      if( currentchar == '\n') { fseek(fileptr, -2, SEEK_CUR); break; }
      else size++;

   }
   char buffer[size]; fread(buffer, 1, size, fileptr);
   printf("Length: %d chars\n", size);
   printf("Buffer: %s\n", buffer);


}


int main(int argc, char *argv[])
{
   if( argc < 2) { printf("Usage: backwards [filename]\n"); return 1; }

   FILE *fileptr = fopen(argv[1], "rb");
   if( fileptr == NULL ) { perror("Error:"); return 1; }

   fseek(fileptr, -1, SEEK_END); /* Seek to END of the file just before EOF */
   read_file(fileptr);


   return 0;


}
```

5.Write a Program using structure to read and display the information about an employee.

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

```
#include <conio.h>

struct details
{
 char name[30];
 int age;
 char address[500];
 float salary;
};

int main()
{
 struct details detail;
 clrscr();
 printf("\nEnter name:\n");
 gets(detail.name);
 printf("\nEnter age:\n");
 scanf("%d",&detail.age);
 printf("\nEnter Address:\n");
 gets(detail.address);
 printf("\nEnter Salary:\n");
 scanf("%f",&detail.salary);


 printf("\n\n\n");
 printf("Name of the Employee : %s \n",detail.name);
 printf("Age of the Employee : %d \n",detail.age);
 printf("Address of the Employee : %s \n",detail.address);
 printf("Salary of the Employee : %f \n",detail.salary);

 getch();
}
```

6.Explain working with files.

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. For Example:

FILE *fp;

To open a file you need to use the fopen function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

FILE *fopen(const char *filename, const char *mode);

Here filename is string literal which you will use to name your file and mode can have one of the following values

w - open for writing (file need not exist)
a - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.
Here's a simple example of using fopen:

```
FILE *fp;

fp=fopen("/home/tutorialspoint/test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")
To close a function you can use the function:

```
int fclose(FILE *a_file);
```

fclose returns zero if the file is closed successfully.
An example of fclose is:

```
fclose(fp);
```

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument.
Try out following example:

```
#include <stdio.h>

main()
{
  FILE *fp;

  fp = fopen("/tmp/test.txt", "w");
  fprintf(fp, "This is testing...\n");
  fclose(fp;);

}
```

Thsi will create a file *test.txt* in /tmp directory and will write *This is testing* in that file.
Here is an example which will be used to read lines from a file:

```
#include <stdio.h>

main()
{
  FILE *fp;
  char buffer[20];

  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buffer);
  printf("Read Buffer: %s\n", %buffer );
  fclose(fp;);

}
```

It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input. The *fgetc* function, which takes a file pointer, and returns an int, will let you read a single character from a file:

int fgetc (FILE *fp);

The *fgetc* returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constnat that indicates that you've reached the end of the file.

The *fputc* function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:

int fputc(int c, FILE *fp);

Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary I/O

There are following two functions which will be used for binary input and output:

```
size_t fread(void *ptr, size_t size_of_elements,
                size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
                size_t number_of_elements, FILE *a_file);
```

Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

7. i)writ a C Program to Add Two Complex Numbers by Passing Structure to a Function

```
#include <stdio.h>
typedef struct complex{
   float real;
   float imag;
}complex;
complex add(complex n1,complex n2);
int main(){
   complex n1,n2,temp;
   printf("For 1st complex number \n");
   printf("Enter real and imaginary respectively:\n");
   scanf("%f%f",&n1.real,&n1.imag);
   printf("\nFor 2nd complex number \n");
   printf("Enter real and imaginary respectively:\n");
   scanf("%f%f",&n2.real,&n2.imag);
   temp=add(n1,n2);
   printf("Sum=%.1f+%.1fi",temp.real,temp.imag);
   return 0;
}
complex add(complex n1,complex n2){
    complex temp;
    temp.real=n1.real+n2.real;
    temp.imag=n1.imag+n2.imag;
    return(temp);
}
```

7 ii)Write a C Program to Calculate Difference Between Two Time Period

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

```
struct TIME{
  int seconds;
  int minutes;
  int hours;
};
void Difference(struct TIME t1, struct TIME t2, struct TIME *diff);
int main(){
    struct TIME t1,t2,diff;
    printf("Enter start time: \n");
    printf("Enter hours, minutes and seconds respectively: ");
    scanf("%d%d%d",&t1.hours,&t1.minutes,&t1.seconds);
    printf("Enter stop time: \n");
    printf("Enter hours, minutes and seconds respectively: ");
    scanf("%d%d%d",&t2.hours,&t2.minutes,&t2.seconds);
    Difference(t1,t2,&diff);
    printf("\nTIME DIFFERENCE: %d:%d:%d - ",t1.hours,t1.minutes,t1.seconds);
    printf("%d:%d:%d ",t2.hours,t2.minutes,t2.seconds);
    printf("= %d:%d:%d\n",diff.hours,diff.minutes,diff.seconds);
    return 0;
}
void Difference(struct TIME t1, struct TIME t2, struct TIME *differ){
    if(t2.seconds>t1.seconds){
        --t1.minutes;
        t1.seconds+=60;
    }
    differ->seconds=t1.seconds-t2.seconds;
    if(t2.minutes>t1.minutes){
        --t1.hours;
        t1.minutes+=60;
    }
    differ->minutes=t1.minutes-t2.minutes;
    differ->hours=t1.hours-t2.hours;
}
```

8.i)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>
int main(){
  char name[50];
  int marks,i,n;
  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);
  return 0;
}
```

8 ii) 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>
struct s
{
char name[50];
int height;
};
int main(){
   struct s a[5],b[5];
   FILE *fptr;
   int i;
   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);
}
```