Lab Program 11. Write a C program to copy a text file to another, read both the input file name and target file name.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   FILE *fptr1, *fptr2;
   char filename[100], c;
   printf("Enter the filename to open for reading \n");
   scanf("%s", filename);
   fptr1 = fopen(filename, "r");
   if (fptr1 == NULL)
   {
      printf("Cannot open file %s \n", filename);
      exit(1);
   }
   printf("Enter the filename to open for writing \n");
   scanf("%s", filename);
   fptr2 = fopen(filename, "w");
   c = getc(fptr1);
   while (c != EOF)
   {
      putc(c, fptr2);
      c = getc(fptr1);
   }
   printf("\nContents copied to %s", filename);
   fclose(fptr1);
   fclose(fptr2);
   return 0;
}
```

Module 5 – Chapter 1: Structure, Union, and Enumerated Data Type

1.1 INTRODUCTION TO STRUCTURES

A structure is a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure. The major difference between a structure and an array is that an array can store only information of same data type.

Arrays Vs Structures

Arrays	**Structures**
Collection of related data items of same type	Collection of elements of different data type
Array is a derived data type	Structures is a user defined data type
To use an array, we must first declare it	To use a structure we must first define and then declare it

1.1.1 Structure Definition & Declaration

A structure is defined using the keyword struct followed by the structure name.

A structure type is generally defined by using the following syntax:

```
struct struct–name
{
        data_type var–name;
        data_type var–name;
        ..............
};
```

For example,

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
};
```

Now the structure has become a user-defined data type. Each variable name declared within a structure is called a member of the structure.

The structure definition (structure template), however, does not allocate any memory or consume storage space.

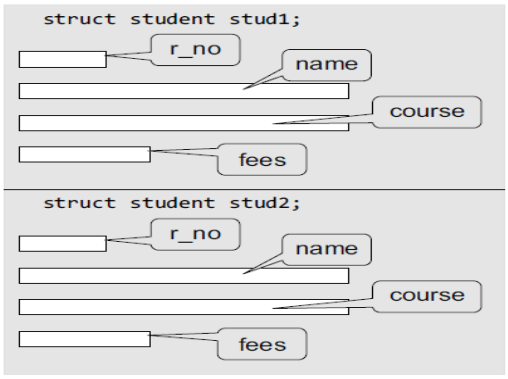
Like any other data type, memory is allocated for the structure when we declare a variable of the structure. For example, we can define a variable of student by writing: **struct student stud1;**

Here, struct student is a data type and stud1 is a variable.

In the following syntax, the variables are declared at the time of structure definition.

```
struct student
{
        int r_no;
        char name[20];
        char course[20];
        float fees;
} stud1, stud2;
```

When we declare variables of the structure, separate memory is allocated for each variable.



Memory allocation for a structure variable

Declare a structure to store customer information.

```
struct customer
{
        int cust_id;
        char name[20];
        char address[20];
};
```

Declare a structure to store information of a particular date.

```
struct date
{
        int day;
        int month;
        int year;
};
```

1.1.2 Accessing the Members of a Structure

A structure member variable is generally accessed using a '.' (dot) operator.

The syntax of accessing a structure or a member of a structure can be given as:

struct_var.member_name

We can access and assign values to the members of a structure in a number of ways. The word title, has no meaning whereas the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'.

For example,

b1.price

is the variable representing the price of **book1** and can be treated like any other ordinary variable. Here is how we would assign values to the members of **book1**.

```
strcpy (b1.title, "BASIC");
strcpy (b1.author, "Balagurusamy");
b1.pages =250;
b1.price =120.50;
```

We can also use **scanf** to give the values through the keyboard.

```
scanf("%s", b1.title);
scanf("%d", &b1.pages);
```

are valid input statements.

The definition and the declaration of structures can be placed either in local or scope when the program has only main() function. But in case the program has more than one function then it is placed in global scope to be accessed by both functions. If it is made local to a function then it can be accessed by only that function.

Program

```
struct  personnel
 {
        char  name[20];
        int   day,year;
        char  month[10];
        float salary;
  };
void main()
{
struct personnel person;
scanf("%s%d%s%d%f",person.name,&person.day,person.month,&person.year,&person.salary);
printf("%s\t%d\t%s\t%d\t%f\n",person.name,person.day,person.month,person.year,person.salary);
}
```

Output

M.L.Goel 10 January 1945 4500.00

M.L.Goel 10 January 1945 4500.00

Program

```
void main()
{
    struct book
    {
        char title[20];
        char author [15];
        int pages;
        float price;
    }b1;

    printf("\n Enter Values");
    scanf("%s%s%d%f",b1.title,b1.author,&b1.pages,&b1.price);
    printf("\n%s\t%s\t%d\t%f",b1.title,b1.author,b1.pages,b1.price);
}
```

Output

```
 Enter Values
Networks balagurusamy 250 550
Networks balagurusamy 250 550
```

Write a program using structures to read and display the information about a student.

```
#include <stdio.h>
int main()
{
struct student
{
int roll_no;
char name[80];
float fees;
char DOB[80];
```

```
};
struct student stud1;
printf("\n Enter the roll number : ");
scanf("%d", &stud1.roll_no);
printf("\n Enter the name : ");
scanf("%s", stud1.name);
printf("\n Enter the fees : ");
scanf("%f", &stud1.fees);
printf("\n Enter the DOB : ");
scanf("%s", stud1.DOB);
printf("\n ROLL No. = %d", stud1.roll_no);
printf("\n NAME = %s", stud1.name);
printf("\n FEES = %f", stud1.fees);
printf("\n DOB = %s", stud1.DOB);
return 0;
}
```

Output

```
Enter the roll number : 01
Enter the name : Rahul
Enter the fees : 45000
Enter the DOB : 25–09–1991
ROLL No. = 01
NAME = Rahul
FEES = 45000.00
DOB = 25–09–1991
```

1.1.3 Initialization of Structures

C language does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of the actual variables. Note that the compile-time initialization of a structure variable must have the following elements:

1.The keyword struct.

2. The structure tag name

3. The name of the variable to be declared.

4. The assignment operator =.

5. A set of values for the members of the structure variable, separated by commas and enclosed in braces

6. A terminating semicolon.

(1)
```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
}b1={“ansi c”,”balaguruswamy”,350,550};
```

(2)
```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
struct book b1={“ansi c”,”balaguruswamy”,350,550};
struct book b2={“basic c”,”balaguruswamy”,350,350};
```

(3) Partial initialization

```
struct book
{
        char title[20];
        char author[15];
        int pages;
        float price;
}b1={"ansi c"};
```

Uninitialized members will be automatically assigned to

- Zero for integer and floating point numbers.
- '\0' for characters and strings

1.1.4 Typedef Declarations

The typedef (derived from type definition) 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 for typedef: **typedef existing datatype new datatype;**

```
typedef int integer;
integer a,b,c;
```

typedef in structures

```
typedef struct student
{
        int rollno;
        char name[20];
}stud;
stud s1,s2,s3;
```

```
struct student
{
        int rollno;
        char name[20];
};
typedef struct student stud;
```

Here stud is the type name. We have not written struct student s1,s2,s3.

Program: use typedef to input and output one employee details consisting employee number and name.

```
void main()
{
        struct employee
        {
                int empno;
                char name[20];
        };
        typedef struct employee emp;
        emp e1;

        printf("\n Enter Values");
        scanf("%d%s”,&e1.empno,e1.name);
        printf("\n%d\t%s”,e1.empno,e1.name);
}
```

Output

```
 Enter Values
201 balagurusamy
201 balagurusamy
```

1.1.5 Copying and Comparing Structures

Two variables of the same structure type can be copied the same way as ordinary variables. If person1 and person2 belong to the same structure called person, then the following statement is valid

person1 = person2;

However, the statements such as

person1 == person2

person1 !=person2

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Program

```
struct student
{
        int  number;
        char name[20];
};
void main()
{
        struct student s1 = {111,"Rao"};
        struct student s2;
        s2 = s1;
        if((s1.number ==  s2.number) &&(strcmp(s1.name,s2.name)==0))
        {
                printf("\nstudent1 and student2 are same\n\n");
         }
        else
                printf("\nstudent1 and student2 are different\n\n");
  }
```

Output

student1 and student2 are same

1.1.6 Operations On Individual Members

The individual members are identified using the member operator, the dot. A member with the dot operator along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators.

We can perform the following operations:

```
student1.marks +10.00;
sum=student1.marks + student2.marks;
```

We can also apply increment and decrement operators to numeric type members. For example, the following statements are valid:

student1.number++;

++ student1.number;

sizeof operator can be used in structures as well. **sizeof(struct student)** will evaluate the number of bytes a structure needs.

```
#include<stdio.h>
struct student
{
        int rollno;
        char name[20];
};

void main()

{

struct student s1;

printf("%d",sizeof(s1));

}
```

Output

22

sizeof(s1) also gives the size of struct student. (22 bytes)

1.2 NESTED STRUCTURES (STRUCTURES WITHIN STRUCTURES)

A structure can be placed within another structure, i.e., a structure may contain another structure as its member. A structure that contains another structure as its member is called a *nested structure*. It means nesting of structures. Also called NESTED STRUCTURES.

It can be implemented in two ways:

First method: Placing the definition of a structure within the definition of another structure.

```
struct student
{
        int rollno;
        char name[20];
        struct dateofbirth
        {
                int day;
                char month[10];
                int year;

        }dob;

}s1;
```

Second method: Placing the declaration (already defined structure) of a structure within the definition of another structure.

```
struct dateofbirth
{
        int day;
        char month[10];
        int year;

};

struct student
{
        int rollno;
        char name[20];
        struct dateofbirth dob;
}s1;
```

The members can be accessed as s1.rollno,s1.name,s1.dob.day,s1.dob.month,s1.dob.year. The innermost structure in a nested structure can be accessed by chaining all the concerned structure variables. We can also nest more than one structure.

Program:

```
void main()
{
        struct student
        {
                int rollno;
                char name[20];
                struct dateofbirth
                {
                        int day;
                        char month[10];
                        int year;

                }dob;

        }s1;

        printf("\n Enter Values");
        scanf("%d%s%d%s%d",&s1.rollno,s1.name,&s1.dob.day,s1.dob.month,&s1.dob.year);
        printf("%d\t%s\t%d\t%s\t%d",s1.rollno,s1.name,s1.dob.day,s1.dob.month,s1.dob.year);
}
```

Output

Enter values

1 neha 16 jan 2010

1 neha 16 jan 2010

1.3 ARRAYS OF STRUCTURES

Let us first analyse where we would need an array of structures.

In a class, we do not have just one student. But there may be at least 30 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures.

The general syntax for declaring an array of structures can be given as,

```
struct struct_name
{
        data_type member_name1;
        data_type member_name2;
        data_type member_name3;
        .....
};
struct struct_name struct_var[index];
```

Consider the given structure definition.

```
struct student
{
int r_no;
char name[20];
char course[20];
float fees;
};
```

A student array can be declared by writing,

```
struct student stud[30];
```

Now, to assign values to the ith student of the class, we will write

```
stud[i].r_no = 09;
strcpy(stud[i].name,"RASHI");
strcpy(stud[i].course,"MCA");
stud[i].fees = 60000;
```

In order to initialize the array of structure variables at the time of declaration, we can write as follows:

```
struct student stud[3] = {{01, "Aman", "BCA", 45000},{02, "Aryan", "BCA", 60000}, {03, "John", "BCA", 45000}};
```

Ex - struct class student [100];

defines an array called student, that consists of 100 elements. Each element is defined to be of the type struct class. Consider the following declaration

```
struct marks
{
        int subject1;
        int subject2;
        int subject3;
};
```

struct marks student [3] ={{45,68,81}, {75,53,69},{57,36,71}};

This declares the student as an array of three elements student[0], student[1], and student[2] and initializes their members as follows:

student [0].subject1= 45;
student[0].subject2=68;
. . . . . .
. . . . . .
student [2].subject3=71;

Lab Program 12. Implement structures to read, write and compute average- marks of the students, list the students scoring above and below the average marks for a class of N students.

Program

```
#include<stdio.h>
struct student
{
  int rollno;
  char name[20];
  float marks;
};
int main()
{
  int i,n;
  struct student s[10];
  float sum=0,average;
```

```
printf("\nEnter the number of student details");
scanf("%d",&n);
for(i=0;i<n;i++)
{
        printf("\nenter the %d student details",i+1);
        printf("\n enter roll number:");
        scanf("%d",&s[i].rollno);
        printf("\n enter student name");
        scanf("%s",s[i].name);
        printf("\n enter the marks:");
        scanf("%f",&s[i].marks);
}
printf("\nStudent details are\n");
printf("\nRollno\t\tName\t\tMarks\n");
for(i=0;i<n;i++)
   printf("%d\t\t%s\t\t%f\n",s[i].rollno,s[i].name,s[i].marks);
for(i=0;i<n;i++)
        sum=sum+s[i].marks;
average=sum/n;
printf("\nAVERAGE=%f",average);
printf("\n students scoring above average\n");
for(i=0;i<n;i++)
{
        if(s[i].marks>=average)
        {
                printf("%s\t",s[i].name);
        }
}
printf("\n students scoring below average\n");
for(i=0;i<n;i++)
{
```

```
        if(s[i].marks<average)
        {
                printf("%s\t",s[i].name);
        }
  }
  return 0;
  }
```

Example program : Array of structures print the names of students who have got more than 75.

Program

```
#include<stdio.h>
struct student
{
  int rollno;
  char name[20];
  float marks;
};
int main()
{
  int i,n;
  struct student s[10];
  printf("\nEnter the number of student details");
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
        printf("\nenter the %d student details",i+1);
        printf("\n enter roll number:");
        scanf("%d",&s[i].rollno);
        printf("\n enter student name");
        scanf("%s",s[i].name);
```

```
        printf("\n enter the marks:");
        scanf("%f",&s[i].marks);
  }
  printf("\nStudent details are\n");
  printf("\nRollno\t\tName\t\tMarks\n");
  for(i=0;i<n;i++)
     printf("%d\t\t%s\t\t%f\n",s[i].rollno,s[i].name,s[i].marks);
  printf("\n students scoring above 75 marks\n");
  for(i=0;i<n;i++)
  {
        if(s[i].marks>=75)
        {
                printf("%s\t",s[i].name);
        }
  }
  return 0;
  }
```

1.4 STRUCTURES AND FUNCTIONS

There are three methods in which we pass a structure to a function as an argument.

1. We can pass each member of a structure. It is not efficient, so we do not use it.
2. We can pass a copy of entire structure to a called function (Call by value)
3. We can pass the entire structure by passing its address (call by reference)

1.4.1 Passing Individual Members

To pass any individual member of a structure to a function, we must use the direct selection operator to refer to the individual members.

```
#include <stdio.h>
typedef struct POINT
{
        int x;
        int y;
```

```
}POINT;
void display(int, int);
int main()
{
        POINT p1 = {2, 3};
        display(p1.x, p1.y);
        return 0;
}
void display(int a, int b)
{
        printf(" The coordinates of the point are: %d %d", a, b);
}
```

Output

The coordinates of the point are: 2 3

1.4.2 Call by Value (Passing the Entire Structure)

In the function call, the structure variable is used. The argument in the function header and declaration is to be declared as a structure variable to receive the entire structure.

Program:

```
        void display(struct student stu);
        struct student
        {
                int rollno;
                char name[20];
        };
        void main()
        {
                struct student s1;
                printf("\n Enter Values");
                scanf("%d%s",&s1.rollno,s1.name);
                display(s1);
```

```
}
void display(struct student stu)
{
        printf("%d\t%s”,stu.rollno,stu.name);
}
```

Output

Enter values

1 neha

1 neha

1.4.3 Call by Reference (Passing Structures through Pointers)

In the function call, the address of the structure variable is used. The argument in the function header and declaration is to be declared as a pointer to a structure variable.

➔ is the member selection operator. (minus sign followed by greater than symbol)

Program:

```
void display(struct student  *stu);
struct student
{
        int rollno;
        char name[20];
};
void main()
{
        struct student s1;
        printf("\n Enter Values");

        scanf("%d%s”,&s1.rollno,s1.name);
        display(&s1);
}
void display(struct student  *stu)
```

```
{
        printf("Roll number=%d\t",stu->rollno);
        printf("Name is %s",stu->name);
}
```

Output

Enter values

1 neha

1.5 SELF-REFERENTIAL STRUCTURES

Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure. For example, consider the structure node given below.

```
struct node
{
        int val;
        struct node *next;
};
```

Here, the structure node will contain two types of data: an integer val and a pointer next.

Actually, self-referential structure is the foundation of other data structures.

Purpose: It is used in linked lists, trees, and graphs.

1.6 UNIONS

Similar to structures, a union is a collection of variables of different data types. The difference between a structure and a union is that in case of unions, we can only store information in one field at any one time.

When a new value is assigned to a field, the existing data is replaced with the new data. Thus, unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time.

1.6.1 Declaring a Union

The syntax for declaring a union is the same as that of declaring a structure. The differences between a structure and a union is that in case of unions, we can only store information in one

field at any one time. Other difference is that instead of using the keyword struct, the keyword union would be used.

The syntax for union declaration can be given as

```
union union–name
{
        data_type var–name;
        data_type var–name;
        .....................
};
```

Again the typedef keyword can be used to simplify the declaration of union variables. The most important thing to remember about a union is that the size of a union is the size of its largest field. This is because sufficient number of bytes must be reserved to store the largest sized field.

1.6.2 Accessing a Member of a Union

A member of a union can be accessed using the same syntax as that of a structure. To access the fields of a union, use the dot operator (.), i.e., the union variable name followed by the dot operator followed by the member name.

1.6.3 Initializing Unions

The difference between a structure and a union is that in case of a union, the fields share the same memory space, so new data replaces any existing data.

```
#include <stdio.h>
typedef struct POINT1
{
        int x, y;
}POINT1;
typedef union POINT2
{
        int x,y;
}POINT2;
int main()
{
        POINT1 P1 = {2,3};
```

```
        // POINT2 P2 ={4,5}; Illegal in case of unions
        POINT2 P2;
        P2.x = 4;
        P2.y = 5;
        printf("\n The coordinates of P1 are %d and %d", P1.x, P1.y);
        printf("\n The coordinates of P2 are %d and %d", P2.x, P2.y);
        return 0;
}
```

Output

The coordinates of P1 are 2 and 3

The coordinates of P2 are 5 and 5

In this code, POINT1 is a structure name and POINT2 is a union name.

However, both the declarations are almost same (except the keywords—struct and union). In main(), we can see the difference between structures and unions while initializing values. The fields of a union cannot be initialized all at once.

For the structure variable the output is as expected but for the union variable the answer does not seem to be correct. To understand the concept of union: The code given below just re-arranges the printf statements.

```
#include <stdio.h>
typedef struct POINT1
{
        int x, y;
}POINT1;
typedef union POINT2
{
        int x,y;
}POINT2;
int main()
{
        POINT1 P1 = {2,3};
        POINT2 P2;
```

```
        printf("\n The coordinates of P1 are %d and %d", P1.x, P1.y);
        P2. x = 4;
        printf("\n The x coordinate of P2 is %d", P2.x);
        P2.y = 5;
        printf("\n The y coordinate of P2 is %d", P2.y);
        return 0;
}
```

Output

The coordinates of P1 are 2 and 3

The x coordinate of P2 is 4

The y coordinate of P2 is 5

1.7 ARRAYS OF UNION VARIABLES

Like structures we can also have an array of union variables. However, because of the problem of new data overwriting existing data in the other fields, the program may not display the accurate results.

```
#include <stdio.h>
union POINT
{
        int x, y;
};
int main()
{
        int i;
        union POINT points[3];
        points[0].x = 2;
        points[0].y = 3;
        points[1].x = 4;
        points[1].y = 5;
        points[2].x = 6;
        points[2].y = 7;
        for(i=0;i<3;i++)
```

```
        printf("\n Coordinates of Point[%d] are %d and %d", i, points[i].x,points[i].y);
        return 0;
}
```

Output

Coordinates of Point[0] are 3 and 3

Coordinates of Point[1] are 5 and 5

Coordinates of Point[2] are 7 and 7

1.8 UNIONS INSIDE STRUCTURES

Generally, unions can be very useful when declared inside a structure. Consider an example in which we want a field of a structure to contain a string or an integer, depending on what the user specifies. The following code illustrates such a scenario:

```
#include <stdio.h>
struct student
{
        union
        {
        char name[20];
        int roll_no;
        };
        int marks;
};
int main()
{
        struct student stud;
        char choice;
        printf("\n You can enter the name or roll number of the student");
        printf("\n Do you want to enter the name? (Y or N): ");
        choice=getchar();
        if(choice=='y' || choice=='Y')
        {
        printf("\n Enter the name:");
```

```
        scanf("%s", stud.name);
        }
        else
        {
        printf("\n Enter the roll number: ");
        scanf("%d", &stud.roll_no);
        }
        printf("\n Enter the marks: ");
        scanf("%d", &stud.marks);
        if(choice=='y' || choice=='Y')
        printf("\n Name: %s ", stud.name);
        else
        printf("\n Roll Number: %d ", stud.roll_no);
        printf("\n Marks: %d", stud.marks);
        return 0;
}
```

Now in this code, we have a union embedded within a structure. We know the fields of a union will share memory, so in the main program we ask the user which data he/she would like to store and depending on his/her choice the appropriate field is used.

1.9 ENUMERATED DATA TYPE

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

```
enum State {Failed = 0, Working = 1};
```

The keyword ‘enum’ is used to declare new enumeration types in C.

enum enumeration_name{identifier$_1$,identifier$_2$,identifier$_3$,….,identifier$_n$};

Example - enum COLORS{RED,BLUE,BLACK,GREEN,YELLOW,PURPLE,WHITE}

After this statement, COLORS has become a new data type. Here, COLORS is the name given to the set of constants. In case we do not assign any value to a constant, the default value for the first one in the list RED (in our case) has the value of 0. The rest of the undefined constants have a value 1 more than its previous one. In our example, RED=0, BLUE=1, BLACK=2, GREEN= 3, YELLOW=4, PURPLE=5, WHITE = 6

Variables of type enum can also be defined. They can be defined in two ways:

1) **enum week{Mon, Tue, Wed};**
   enum week day;
2) **enum week{Mon, Tue, Wed}day;**

```
// An example program to demonstrate working of enum in C
#include<stdio.h>
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
 int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Output:
2

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Another example of enumeration is:

```
// Another example program to demonstrate working of enum in C
#include<stdio.h>
 enum year{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
 int main()
{
   enum year i;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);
   return 0;
}
```

Output:

0 1 2 3 4 5 6 7 8 9 10 11

In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

Interesting facts about initialization of enum.

1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};
 int main()
{
  printf("%d, %d, %d", Working, Failed, Freezed);
  return 0;
}
```

Output:
1, 0, 0

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday, saturday};

int main()
{
   enum day d = thursday;
   printf("The day number stored in d is %d", d);
   return 0;
}
```

Output:
The day number stored in d is 4

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

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

```
enum day {sunday = 1, monday, tuesday = 5,
        wednesday, thursday = 10, friday, saturday};

int main()
{
   printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
         wednesday, thursday, friday, saturday);
   return 0;
}
```

Output:
1 2 5 6 10 11 12

4. The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};
 int main()  {
return 0;
}
```

Output:

Compile Error: 'failed' has a previous declaration as 'state failed'

fseek() function in files

fseek() is used to move the file pointer associated with a given file to a specific position.

The fseek() syntax is:

int **fseek**(FILE ***pointer*, long int *offset*, int *position*);

- **file pointer:** It is the pointer to a FILE object.
- **offset:** It is the number of bytes to offset from the position

- **position:** It is the position from where the offset is added. Position defines the point with respect to which the file pointer needs to be moved. It has three values:
    - **SEEK_END**: It denotes the end of the file.
    - **SEEK_SET**: It denotes starting of the file.
    - **SEEK_CUR**: It denotes the file pointer's current position.
- It returns zero if successful, or else it returns a non-zero value.

```
#include <stdio.h>

int main()

{

   FILE* fp;

   fp = fopen("test.txt", "r");

    fseek(fp, 0, SEEK_CUR);

    // Printing position of pointer

   printf("%d", ftell(fp));

    return 0;

}
```

Output

0

Structure	***Union***
Keyword: struct	Keyword: union
We can store information all fields at any one time.	We can only store information in one field at any one time.
Memory allocated for all the fields.	Size of a union is the size of its largest field.

struct structure_name
{
datatype member$_1$;
datatype member$_2$;
.
.
datatype member$_n$;
 	
};
	union union_name
	{
	datatype member$_1$;
	datatype member$_2$;
	.
	.
	datatype member$_n$;
	
	};

struct student
{
int rollno;
char name[20];
};
Memory allocated: 22 bytes
	union student
	{
	int rollno;
	char name[20];
	};
	Memory allocated: 20 bytes

program	program

MODULE – 5

Structure, Union, and Enumerated Data Type: Introduction, structures and functions, Unions, unions inside structures, Enumerated data type.

Files: Introduction to files, using files in C, reading and writing data files, Detecting end of file

Textbook: Chapter 15.1 – 15.10, 16.1-16.5

CHAPTER 2 FILES

2.1 INTRODUCTION TO FILES

- A file is a collection of data stored on a secondary storage device like hard disk.
- Till now, we had been processing data that was entered through the computer's keyboard.
- But this task can become very tedious especially when there is a huge amount of data to be processed.
- A better solution, therefore, is to combine all the input data into a file and then design a C program to read this data from the file whenever required.

The console-oriented I/O operations pose two major problems:

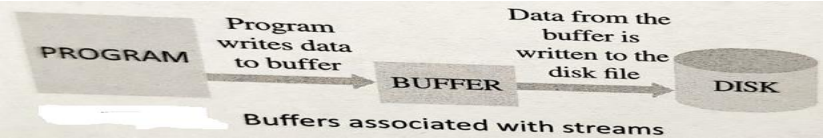
1. First, it becomes cumbersome and time-consuming to handle huge amount of data through terminals.
2. Second, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off. Therefore, it becomes necessary to store data on a permanent storage device (e.g. hard disks) and read whenever required, without destroying the data.

- In order to use files, we have to learn file input and output operations, i.e., how data is read from or written to a file.
- Although file I/O operations are almost same as terminal I/O, the only difference is that when doing file I/O, the user must specify the name of the file from which data should be read/written.

2.1.1 Buffer Associated with files

A buffer is nothing but a block of memory that is used for temporary storage of data that has to be read from or written to a file.

The creation and operation the buffer is automatically handled by the operating system. However, C provides some functions for buffer manipulation. The data resides in the buffer until the buffer is flushed or written to a file.



2.1.2 Types of Files

In C, the types of files used can be broadly classified into two categories ASCII text files and binary files.

ASCII Text Files

The file that contains ASCII codes of data like digits, alphabets and symbols is called text file (or) ASCII file.

Because text files only process characters, they can only read or write data one character at a time. A line in a text file is not a C string, so it is not terminated by a null character.

The contents of a binary file are not human-readable. If we want the data stored in the file to be human-readable, then store the data in a text file. In a text file, each line of data ends with a newline character. Each file ends with a special character called the end-of-file (EOF) marker.

Binary Files

A binary file may contain any type of data, encoded in binary form for computer storage and processing purposes.

The file that contains data in the form of bytes (0's and 1's) is called as binary file. Generally, the binary files are compiled version of text files. Like text file, binary file also ends with an EOF marker.

2.2 USING FILES IN C

To use files in C, we must follow the steps given below:

- declare a file pointer variable
- open the file
- process the file (Reading from a file and Writing to a file)
- close the file

2.2.1 Declaring a File Pointer Variable

There can be a number of files on the disk. In order to access a particular file, we must specify the name of the file that has to be used. This is accomplished by using a file pointer variable that points to a structure FILE (defined in stdio.h).

The syntax for declaring a file pointer is FILE *file_pointer_name;

FILE *fp;

Then, fp is declared as a file pointer.

2.2.2 Opening a File

A file must be opened before any operation can be performed on it. A file must first be opened before data can be read from it or written to it. In order to open a file, the fopen() function is used. The prototype of fopen() can be given as

```
FILE *file_pointer_name = fopen ("file_name", "Mode");
```

pointer_name can be anything of our choice.

file_name is the name of the file, which we want to open. While opening a file, we need to specify the mode.

Example - FILE * fp;

fp = fopen("Student.txt", "r");

Using the above declaration, the file whose pathname is the string pointed to by file name is opened in the mode specified using the mode. If successful, fopen() returns a pointer-to-file and if it fails, it returns NULL.

File Name

In C, fopen() may contain the path information instead of specifying the filename. The path gives information about the location of the file on the disk. If a filename is specified without a path, it is assumed that the file is located in the current working directory. For example, if a file named student.txt is located on D drive in directory BCA, then the path of the file can be specified by writing, D:\BCA\student.txt

In C, a backslash character has a special meaning with respect to escape sequences when placed in a string. So in order to represent a backslash character in a C program, we must precede it with another backslash. Hence, the above path will be specified as given below in the C program.

D:\\BCA\\student.txt

File Mode

The second argument in fopen() is the mode. Mode conveys to C the type of processing that will be done with the file.

Sl.No	Mode	Description
1	r	Opens a text file in **reading** mode. Only reading possible. Does not create the file if it does not exist.
2	w	Only writing is possible. Create the file if it does not exist; otherwise, erase the old content of the file and open a blank file.
3	a	Only writing is possible. Create a file; if it does not exist, otherwise open the file and write from the end of the file. (Does not erase the old content).
4	r+	Reading and writing are possible. Create a file if it does not exist, overwriting existing data. Used for modifying content.
5	w+	Reading and writing are possible. Create a file if it does not exist. Erase old content.
6	a+	Reading and writing are possible. Create a file if it does not exist. Append content at the end of the file.

The above modes are used with text files only. If we want to work with binary files we use **rb, wb, ab, rb+, wb+ and ab+**.

Example Code -

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
        FILE *fp=fopen("C:\\Users\\sonim\\Documents\\zoom\\stu.txt","r");
        if(fp==NULL)
        {
        printf("\n The file could not be opened");
        exit(1);
        }
}
```

The fopen() function can fail to open the specified file if we attempt to open a non-existent file for reading.

2.2.3 Closing a File Using fclose()

To close an open file, the fclose() function is used which disconnects a file pointer from a file. After fclose() has disconnected the file pointer from the file, the pointer can be used to access a different file or the same file but in a different mode. The fclose() function not only closes the tile, but also flushes all the buffers that are maintained for that file. If we do not close a file after using it, the system closes it automatically when the program exits. However, since there is a limit on the number of files which can be opened simultaneously, we must close a file after it is used.

Syntax – int fclose(FILE *fp);

Example – fclose(fp);

fp is the file pointer which points to the file that has to be closed. The function returns an integer value which indicates fclose() was successful or not. A zero is returned if the function was successful and a non-zero is returned if an error occurred.

If a file's buffer has to be flushed without closing it then use fflush() function.

2.3 READING DATA FROM FILES

The reading from a file operation is performed using the following pre-defined file handling methods.

1. getc()
2. getw()
3. fscanf()
4. fgets()
5. fread()

1. **getc(*file_pointer) -** This function is used to read a character from specified file which is opened in reading mode. It reads from the current position of the cursor. After reading the character the cursor will be at next character.

```
#include<stdio.h>

int main()

{

  FILE *fp;
```

```
    char ch;
    fp = fopen("MySample.txt","r");
    ch = getc(fp);
    printf("ch = %c", ch);
    fclose(fp);
    return 0;
}
```

2. **getw(*file_pointer) -** This function is used to read an integer value form the specified file which is opened in reading mode. If the data in file is set of characters then it reads ASCII values of those characters.

```
#include<stdio.h>
int main()
{
    FILE *fp;
    int i,j;
    fp = fopen("MySample.txt","w");
    putw(65,fp);  // inserts A
    putw(97,fp);  // inserts a
    fclose(fp);
    fp = fopen("MySample.txt","r");
    i = getw(fp);   // reads 65 - ASCII value of A
    j = getw(fp);   // reads 97 - ASCII value of a
    printf("SUM of the integer values stored in file = %d", i+j);  // 65 + 97 = 162
    fclose(fp);
```

```
    return 0;
}
```

3. **fscanf(*file_pointer, typeSpecifier, &variableName) -** This function is used to read multiple datatype values from specified file which is opened in reading mode.

```
#include<stdio.h>
int main()
{
  char str1[10], str2[10], str3[10];
  int year;
  FILE * fp;
  fp = fopen ("file.txt", "w+");
  fputs("We are in 2016", fp);
rewind(fp);    // moves the cursor to begining of the file
fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
printf("Read String1 - %s\n", str1);
printf("Read String2 - %s\n", str2);
printf("Read String3 - %s\n", str3);
printf("Read Integer - %d", year);
fclose(fp);
return 0;
}
```

4. **fgets(variableName, numberOfCharacters, *file_pointer) -** This method is used for reading a set of characters from a file which is opened in reading mode starting from the

current cursor position. The fgets() function reading terminates with reading NULL character.

```
#include<stdio.h>
int main()
{
        FILE *fp;
        char str[20];
        fp = fopen ("file.txt", "r");
        fgets(str,6,fp);
        printf("str = %s", str);
        fclose(fp);
        return 0;
}
```

5. **fread(source, sizeofReadingElement, numberOfCharacters, FILE *pointer) -** This function is used to read specific number of sequence of characters from the specified file which is opened in reading mode.

```
#include<stdio.h>
#include<string.h>
int main()
{
  FILE *fp;
  char str[20];
  fp = fopen ("file.txt", "r");
  fread(str,sizeof(char),5,fp);
  printf("str = %s", str);
  fclose(fp);
  return 0;
```

```
}
```

2.4 WRITING DATA TO FILES

The writing into a file operation is performed using the following pre-defined file handling methods.

1. putc()
2. putw()
3. fprintf()
4. fputs()
5. fwrite()

1. **putc(char, *file_pointer) -** This function is used to write/insert a character to the specified file when the file is opened in writing mode.

```
#include<stdio.h>
int main(){
  FILE *fp;
  char ch;
  fp = fopen("MySample.txt","w");
  putc('A',fp);
  ch = 'B';
  putc(ch,fp);
  fclose(fp);
  return 0;
}
```

2. **putw(int, *file_pointer) -** This function is used to writes/inserts an integer value to the specified file when the file is opened in writing mode.

   Same Program under getw()

```
#include<stdio.h>
int main()
{
  FILE *fp;
  int i,j;
  fp = fopen("MySample.txt","w");
  putw(65,fp);  // inserts A
  putw(97,fp);  // inserts a
  fclose(fp);
  fp = fopen("MySample.txt","r");
  i = getw(fp);   // reads 65 - ASCII value of A
  j = getw(fp);   // reads 97 - ASCII value of a
  printf("SUM of the integer values stored in file = %d", i+j);  // 65 + 97 = 162
  fclose(fp);
  return 0;
}
```

3. **fprintf(*file_pointer, "text") -** This function is used to writes/inserts multiple lines of text with mixed data types (char, int, float, double) into specified file which is opened in writing mode.

```
  #include<stdio.h>
  int main()
  {
  FILE *fp;
  char *text = "\nthis is example text";
```

```
int i = 10;
fp = fopen("MySample.txt","w");
fprintf(fp,"This is line1\nThis is line2\n%d", i);
fprintf(fp,text);
fclose(fp);
return 0;
}
```

4. **fputs("string", *file_pointer) -** TThis method is used to insert string data into specified file which is opened in writing mode.

```
#include<stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("MySample.txt","w");
  fputs("Hi!\nHow are you?",fp);
  fclose(fp);
  return 0;
}
```

5. **fwrite("StringData", sizeof(char), numberOfCharacters, FILE *pointer) -** This function is used to insert specified number of characters into a file which is opened in writing mode.

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

```
int main()
{
  FILE *fp;
  char *text = "Welcome to C Language";
  fp = fopen("MySample.txt","w");
  fwrite(text,sizeof(char),5,fp);
  fclose(fp);
  return 0;
}
```

2.5 DETECTING THE END-OF-FILE

When reading or writing data to files, we often do not know exactly how long the file is. For example, while reading the file, we usually start reading from the beginning and proceed towards the end of the file. In C, there are two ways to detect the end of file.

1. EOF
2. feof

EOF

While reading the file, character by character, the programmer can compare the character that has been read with EOF, which is a symbolic constant defined in stdio.h.

Let's say we have "new.txt" file with the following content.

This is demo!

This is demo!

Example

```
#include <stdio.h>
int main() {
  FILE *f = fopen("new.txt", "r");
```

```
  int c = getc(f);
  while (c != EOF) {
    putchar(c);
    c = getc(f);
  }
  fclose(f);
  return 0;
}
```

Output

This is demo!

This is demo!

In the above program, file is opened by using fopen(). When integer variable c is not equal to EOF, it will read the file.

feof()

We can use the standard library function feof() which is defined in stdio.h.

The function feof() takes the file pointer as an argument and returns zero (false) when the end of file has not been reached and a one (true) if the end of file has been reached.

Syntax of feof() - int feof(FILE *file_pointer)

Example – Let's say we have "new.txt" file with the following content.

This is demo!

This is demo!

```
#include <stdio.h>

#include<stdlib.h>

void main()
```

```
{
  FILE *fp = fopen("new.txt", "r");
  char str[100];
  if(fp==NULL)
  {
    printf("The file could not be opened");
   exit(1);
  }
  while(1)
  {
        fgets(str,79,fp);
        if(feof(fp))
              break;
        printf("%s",str);
  }
  fclose(fp);
}
```

Output

This is demo!

This is demo!

The function feof() is checking that pointer has reached to the end of file or not.