Course: Problem Solving Techniques using C

Course Code: 24BTPHY104

Module 5

5. STRUCTURES AND UNIONS

- Structures and unions are two of many user defined data types in C.
- Both are similar to each other but there are quite some significant differences.

5.1 Structures

- A structure simply can be defined as a user-defined data type which groups logically related items under one single unit. We can use all the different data items by accessing that single unit.
- All the data items are stored in contiguous memory locations.
- It is not only permitted to one single data type items and it can store items of different data items.

5.1.1 Declaring a Structure

```
struct structure_name
{
data_type variable_name;
data_type variable_name;
........
data_type variable_name;
};
```

Example 1:

```
struct student
{
   char  name[100];
   int   roll;
   float marks;
};
```

Example 2:

Suppose we need to store the details of an employee. It includes ID, name, age, salary, designation and several other factors which belong to different data types.

```
struct Empl
{
int  emp_id;
float salary;
char designation[20];
int depart_no;
int age_of_emp;
};
```

5.1.2 Initialization of Structure

Like any other data type, structure variable can also be initialized at compile time.

```
struct Patient
{
float height;
int weight;
int age;
};
struct Patient p1 = { 180.75 , 73, 23 }; //initialization
```

[OR]

```
struct Patient p1;
p1.height = 180.75; //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Example :

C Program to declare a structure student with field name and roll no. take input from user and display them.

```
#include<stdio.h>
struct student
{
char name[10];
```

```
int roll;
};
void main()
{
struct student s1;
clrscr();
printf("\n Enter student record\n");
printf("\n student name\t");
scanf("%s",s1.name);
printf("\nEnter student roll\t");
scanf("%d",&s1.roll);
printf("\nstudent name is %s",s1.name);
printf("\nroll is %d",s1.roll);
getch( );
}
```

OUTPUT:

```
Enter student record
Student name yaswanth
Enter student roll 35

Student name is yaswanth
roll is 35
```

5.2 Unions

- A union is a user-defined data type similar to a structure, but with a key difference: all members of a union share the same memory location. This means only one member can hold a value at any given time.
- The size of the union is determined by the size of its largest member.
- Unions are useful for memory management and situations where a variable may take on different types of values at different times.

Syntax of union

```
union UnionName {
```

```
    dataType member1;
    dataType member2;
    // more members
};
```

5.2.1 Declaring Union

```
union Data {
    int i;
    float f;
    char str[20];
};
```

5.2.1 Initialization and Usage

```
union Data data;
data.i = 10;
printf("data.i: %d\n", data.i);

data.f = 220.5;
printf("data.f: %.2f\n", data.f);

strcpy(data.str, "C Programming");
printf("data.str: %s\n", data.str);
```

Defining of Union

- A union has to defined, before it can be used.
- The syntax of defining a union is

```
union <union_name>
{
   <data_type> <variable_name>;
   <data_type> <variable_name>;
 ………….
 <data_type> <variable_name>;
 } ;
```

Example:

To define a simple union of a char variable and an integer variable

```
union shared
{
char c;
int i;
} ;
```

This union, **shared**, can be used to create instances of a union that can hold either a character value(c) or an integer value(i).

Union Data Type

- A union is a user defined data type like structure.
- The union groups logically related variables into a single unit.
- The union data type allocates the space equal to space needed to hold the largest data member of union.
- The union allows different types of variable to share same space in memory.
- The method to declare, use and access the union is same as structure.

Difference between Structures and Union

- **Memory Allocation:**
  - **Structures:** Each member has its own memory location. The total size of the structure is the sum of the sizes of all members.
  - **Unions:** All members share the same memory location. The size of the union is the size of its largest member.
- **Usage:**
  - **Structures:** Used when you need to store multiple related data items together.
  - **Unions:** Used when you need to store one of several possible data items in the same memory location, optimizing for memory usage.

5.3 Nested Structures in C

A **nested structure** in C is a structure within structure.

Syntax of Nested Structure in C

```
struct outer_struct
{
```

```
    // outer structure members
    int outer_member1;
  float outer_member2;
  // nested structure definition
   struct inner_struct
   {
    // inner structure members
    int inner_member1;
    float inner_member2;
  } inner;
};
```

Example of Nested Structure in C:

```
struct school
{
  int numberOfStudents;
  int numberOfTeachers;
   struct student{
      char name[50];
      int class;
      int roll_Number;
   } std;
};
```

In the above example of nested structure in C, there are two structures **Student (depended structure)** and another structure called **School(Outer structure)**.

- The structure School has the data members like numberOfStudents, numberOfTeachers, and the Student structure is nested inside the structure School and it has the data members like name, class, roll_Number.
- To access the members of the school structure, you use dot notation.
- For example, to access the numberOfStudents member of the school structure, you would use the following syntax:

```
struct school s;
s.numberOfStudents = 100;
```

- To access the members of the nested student structure, you would use the following syntax:
  - strcpy(s.std.name, "John Smith");
  - s.std.class = 10;
  - s.std.roll_Number = 1;
- Note that the dot notation is used to access the members of the std structure, which is a member of the school structure.
- The nested structure student contains three members: name, class, and roll_Number.
- This structure can be useful in situations where you need to store information about a student, such as their name, class, and roll number.
- The structure can be nested in the following ways.

1. Separate structure
2. Embedded structure

1. Separate structure

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

- doj (date of joining) is the variable of type Date.
- Here doj is used as a member in Employee structure.

Embedded structure

- The embedded structure enables us to declare the structure inside the structure.
- Hence, it requires less line of codes but it cannot be used in multiple data structures.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
     int dd;
     int mm;
     int yyyy;
   }doj;
}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

```
e1.doj.dd
e1.doj.mm
e1.doj.yyyy
```

5.4 Arrays of Structures in C

- An array whose elements are of type structure is called array of structure.
- It is generally useful when we need multiple structure variables in our program.

- To create an array of structure, first structure is declared and then array of structure is declared just like an ordinary array.

Example: (Array of Structure)

```
struct employee
{
 int emp_id;
 char name[20];
 char dept[20];
 float salary;
};
```

- An array of structure can be created as follows:

  struct employee emp[10]; /* This is array of structure */

- In this example, the first structure employee is declared, then the array of structure created using a new type i.e. struct employee.
- Using the above array of structure, 10 set of employee records can be stored and manipulated.

Accessing Elements from Array of Structure

- To access any structure, index is used.
- For example, to read the emp_id of the first structure we use

  scanf("%d", emp[0].emp_id);

- In C, array indexing starts from 0.
- Similar array of structure can also be declared as follows:

```
struct employee
{
 int emp_id;
 char name[20];
 char dept[20];
 float salary;
```

```
}emp[10];
```

- The **typedef** can be used to create similar array of structure as:

```
typedef     struct
{
int emp_id;
char name[20];
char dept[20];
float salary;
}employee;
employee emp[10];
```

Example:

C program to read records of three different students in structure having member name, roll and marks, and displaying it.

```
#include<stdio.h>
/* Declaration of structure */
struct student
{
char name[30];
int roll;
float marks;
};
int main()
{
 /* Declaration of array of structure */
 struct student s[3];
 int i;
 for(i=0;i<3;i++)
 {
  printf("Enter name, roll and marks of student:\n");
```

```
  scanf("%s%d%f",s[i].name, &s[i].roll, &s[i].marks);
 }
 printf("Inputted details are:\n");
 for(i=0;i<3;i++)
 {
  printf("Name: %s\n",s[i].name);
  printf("Roll: %d\n", s[i].roll);
  printf("Marks: %0.2f\n\n", s[i].marks);
 }
 return 0;
}
```

OUTPUT:

```
Enter name, roll and marks of student:
Gopinath
17
80.5
Enter name, roll and marks of student:
Mohan
18
90.00
Enter name, roll and marks of student:
Manjula
19
78.00
Inputted details are:
Name: Gopinath
Roll: 17
Marks: 80.50

Name: Mohan
Roll: 18
Marks: 90.00
```

Name: Manjula

Roll: 19

Marks: 78.00

5.5 Passing structures to functions

Structures can be passed as arguments to the functions. This can be done in three ways. They are,

- Passing the members of the structures as an argument.
- Passing the entire structure as an argument.
- Passing the address of the structure as arguments.

Pass Structure Members (Variables) to Functions

- Sometimes we don’t want to pass the entire structure to the function.
- We want to pass only a few members of the structure.
- We can use the dot (.) operator to access the individual members of the structure and pass them to the function.

Example:

- Let us create a structure to hold the details of a student, such as the name of the student, roll number, and marks.
- We want to print only the roll number and marks using a function.
- In this case, passing the entire structure to the function is unnecessary when we want to print only a few structure members.

// C Program to pass structure members to functions

```
#include <stdio.h>
struct student
{
   char name[50];
   int per, rno;   // declare percentage and roll number as
```

```
                    // integer data type
};
void display(int a, int b);
int main()
{
  struct student s1;
  printf("Enter name: ");
   scanf("%s",&s1.name);
  printf("Enter the roll number: ");
  scanf("%d",&s1.rno);
  printf("Enter percentage: ");
  scanf("%d", &s1.per);
  display(s1.rno,s1.per);
  return 0;
}
void display(int a, int b)
{
  printf("\nDisplaying information\n");
  printf("Roll number: %d", a);
  printf("\nPercentage: %d", b);
}
```

- ✓ In the above example, we created a structure to hold the name, roll number, and percentage of the student.
- ✓ The input from the user is stored in the structure.
- ✓ A function named display() is created, which takes the roll number and the percentage of the student as the parameter.
- ✓ Using the dot (.) operator, we accessed the member of the structure and passed it to the function.

OUTPUT:

The output of the above code is as follows:

Enter name: Shankar

Enter the roll number: 42

Enter percentage: 98

Displaying information

Roll number: 42

Percentage: 98

Pass Structure by Reference (Address)

- ✓ Passing the parameter as a value will make a copy of the structure variable, passing it to the function.
- ✓ Imagine we have a structure with a huge number of structure members.
- ✓ Making a copy of all the members and passing it to the function takes a lot of time and consumes a lot of memory.
- ✓ To overcome this problem, we can pass the address of the structure.
- ✓ Pointers are the variables that hold the address of other variables.
- ✓ We can use pointers to pass the structure by reference.

Example:

// C program to pass the structure by reference

```
#include<stdio.h>
struct car
{
   char name[20];
   int seat;
   char fuel[10];
};
void print_struct(struct car *);
```

```
int main()
{
    struct car tata;
    printf("Enter the model name : ");
    scanf("%s",tata.name);
    printf("\nEnter the seating capacity : ");
    scanf("%d",&tata.seat);
    printf("\nEnter the fuel type : ");
    scanf("%s",tata.fuel);
    print_struct(&tata);
    return 0;
}
void print_struct(struct car *ptr)
{
    printf("\n---Details---\n");
    printf("Name: %s\n", ptr->name);
    printf("Seat: %d\n", ptr->seat);
    printf("Fuel type: %s\n", ptr->fuel);
    printf("\n");
}
```

Explanation:

- In the above code, a structure named car and a function named print_struct() are defined.
- The structure stores the model name, seating capacity, and the fuel type of the vehicle.
- In the main() function, we created a structure variable named tata and stored the values.
- Later the address of the structure is passed into the print_struct() function, which prints the details entered by the user.
- The address is passed using the address operator ampersand (&).
- To access the pointer members, we use the arrow operator -> operator.

OUTPUT:

The output of the above code is as follows:

Enter the model name : ALtroz

Enter the seating capacity : 5

Enter the fuel type : Petrol

---Details---

Name: ALtroz

Seat: 5

Fuel type: Petrol

5.6 typedef in C

- The **typedef** is a keyword that is used to provide existing data types with a new name.
- The C typedef keyword is used to redefine the name of already existing data types.
- When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.

Syntax of typedef

typedef <existing_name> <alias_name>

- In the above syntax, '**existing_name'** is the name of an already existing variable while '**alias name'** is another name given to the existing variable.
- For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type.
- To overcome the problem, we use **a typedef** keyword.
- **typedef** unsigned **int** unit;
- In the above statements, we have declared the **unit** variable of type unsigned int by using **a typedef** keyword.
- we can create the variables of type **unsigned int** by writing the following statement:
- unit a, b; instead of writing
    - unsigned **int** a, b;
- the **typedef** keyword provides a shortcut by providing an alternative name for an already existing variable.

- This keyword is useful when we are dealing with the long data type especially, structure declarations.

Example 1 (typedef):

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
j=20;
printf("Value of i is: %d",i);
printf("\nValue of j is: %d",j);
return 0;
}
```

OUTPUT:

Value of i is :10

Value of j is :20

Example 2: (typedef using structures)

```
struct student
{
char name[20];
int age;
};
struct student s1;
```

- In the above structure declaration, we have created the variable of student type by writing the following statement:
    - struct student s1;
- The above statement shows the creation of a variable, i.e., s1, but the statement is quite big.

- To avoid such a big statement, we use the typedef keyword to create the variable of type student.

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

- In the above statement, we have declared the variable stud of type struct student.
- Now, we can use the stud variable in a program to create the variables of type struct student.
- The above typedef can be written as:
  - typedef struct student
  - {
  - char name[20];
  - int age;
  - } stud;
  - stud s1,s2;
- The above declarations show that the typedef keyword reduces the length of the code and complexity of data types.

5.7 Enum (Enumerated Data type) in C

- The enum in C is also known as the enumerated type.
- It is a user-defined data type that consists of integer values, and it provides meaningful names to these values.
- The use of enum in C makes the program easy to understand and maintain.
- The enum is defined by using the enum keyword.

Defining the enum:

enum flag{const1, const2,.....constN};

- In the above declaration, we define the enum named as flag containing 'N' integer constants.
- The default value of integer_const1 is 0, integer_const2 is 1, and so on.
- We can also change the default value of the integer constants at the time of the declaration.

Example:

enum fruits{mango, apple, strawberry, papaya};

- The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3.
- The default value can be changed by assigning different values as follows:

```
enum fruits
{
mango=2,
apple=1,
strawberry=5,
papaya=7,
};
```

Declaration of enum data type

enum boolean {false, true};

enum boolean check; // declaring an enum variable

- A variable **check** of the type **enum boolean** is created.
- The enum variables can also be declared as follows:
    - **enum boolean {false, true} check;**
- The value of false is equal to 0 and the value of true is equal to 1.

Example: (Enumeration type)

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

```
int main()
{
   // creating today variable of enum week type
   enum week today;
   today = Wednesday;
   printf("Day %d",today+2);
   return 0;
}
```

OUTPUT:

Day 5

5.8 Bit Fields in C

- In C, we can specify the size (in bits) of the structure and union members.
- The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.
- Bit fields in C language are used when the storage of our program is limited.

Need of Bit Fields in C

- Reduces memory consumption.
- To make our program more efficient and flexible.
- Easy to Implement.

Declaration of C Bit Fields

Bit-fields are variables that are defined using a predefined width or size.

Syntax of Bit Fields in C

```
struct
{
data_type member_name : width_of_bit-field;
};
```

where,

- **data_type:** It is an integer type that determines the bit-field value which is to be interpreted. The type may be int, signed int, or unsigned int.
- **member_name:** The member name is the name of the bit field.
- **width_of_bit-field:** The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Example: (Bitfields)

Consider the following example code.

```
struct Date
{
    unsigned int day;
    unsigned int month;
    unsigned int year;
} ;
```

- Here, the variable of Date structure allocates 6 bytes of memory.
- In the above example structure, the members day and month both does not requires 2 bytes of memory for each.
- Because member day stores values from 1 to 31 only which requires 5 bits of memory, and the member month stores values from 1 to 12 only which required 4 bits of memory.
- So, to save the memory, we use the bitfields.
- Consider the following structure with bitfields.

```
struct Date
{
unsigned int day : 5;
unsigned int month : 4;
unsigned int year;
} ;
```

- Here, the variable of Date structure allocates 4 bytes of memory.

Applications of C Bit Fields

1.Limited Storage: Bit-fields are a suitable choice when you have limited memory or storage resources, as they allow you to represent and manipulate data in a compact way, using fewer bits than standard data types.

2.Device Status and Information: In situations where devices need to transmit status or information encoded as multiple bits, bit-fields are an efficient means of representing this data, optimizing the use of available bits and ensuring efficient communication.

3.Encryption Routines: Encryption algorithms often work at the bit level to manipulate and transform data.

Bit-fields can be a valuable tool in encryption routines to access and manipulate specific bits within a byte, making them useful for bitwise operations and encryption-related tasks.

5.9 Command Line arguments

- ✓ In C programming language, command line arguments are the data values that are passed from command line to our program.
- ✓ When the main function of a program contains arguments, then these arguments are known as Command Line Arguments.
- ✓ The main function can be created with two methods: first with no parameters (void) and second with two parameters.
- ✓ The parameters are argc and argv, where argc is an integer and the argv is a list of command line arguments.
- ✓ argc denotes the number of arguments given, while argv[] is a pointer array pointing to each parameter passed to the program. If no argument is given, the value of argc will be 1.
- ✓ The value of argc should be non-negative.
- ✓ Using command line arguments, we can control the program execution from the outside of the program.
- ✓ Generally, all the command line arguments are handled by the main() method. Generally, the command line arguments can be understood as follows.

✓ When command line arguments are passed main() method receives them with the help of two formal parameters and they are:

- **int argc**
- **char *argv[ ]**

- ***int argc -*** It is an integer argument used to store the count of command line arguments are passed from the command line.
- ***char *argv[ ] -*** It is a character pointer array used to store the actual values of command line arguments are passed from the command line.

Example:

// C Program to illustrate command line arguments

```
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[])
{
int i;
  if(argc == 1)
{
        printf("Please provide command line arguments!!!");
        return 0;
   }
   else
   {
     printf("Total number of arguments are - %d and  they are\n\n", argc);
        for(i=0; i<argc ; i++)
       {
            printf("%d -- %s \n", i+1, argv[i]);
        }
        return 0;
   }
}
```

5.10 C pre-processor directives

The important functions of a preprocessor are to include the header files that contain the library functions used in the program.

A preprocessor section of the program always appears at the top of the C code.

Each preprocessor statement starts with the hash (#) symbol.

List of Preprocessor Directives

To execute a preprocessor program on a certain statement, some of the preprocessor directives types are:

- **#define:** It substitutes a preprocessor using macro.
- **#include:** It helps to insert a certain header from another file.
- **#undef:** It undefines a certain preprocessor macro.
- **#ifdef:** It returns true if a certain macro is defined.
- **#ifndef:** It returns true if a certain macro is not defined.
- **#if, #elif, #else, and #endif:** It tests the program using a certain condition; these directives can also be nested.
- **#line:** It handles the line numbers on the errors and warnings. It can be used to change the line number and source files while generating output during compile time.
- **#error and #warning:** It can be used for generating errors and warnings.
    1. #error can be performed to stop compilation.
    2. #warning is performed to continue compilation with messages in the console window.
- **#region and #endregion:** To define the sections of the code to make them more understandable and readable, we can use the region using expansion and collapse features.
- **#pragma:** Issues special commands to the compiler, using a standardized method.

Example:

#define MAX_ARRAY_LENGTH 20

- This **#define** directive tells the CPP to replace the instances of MAX_ARRAY_LENGTH with 20.
    - **#include** <stdio.h>
- #include directive tell the compiler to get "**stdio.h**" from the System Libraries and add the text to the current source file.
    - **#include** "myheader.h"
- #include tells compiler to get "myheader.h" from the local directory and add the content to the current source file.
    - #undef FILE_SIZE
    - #define FILE_SIZE 45

- #undef directive tell the compiler to undefine existing FILE_SIZE and define it as 45.
    - #ifndef MESSAGE
    - #define MESSAGE "You wish!"
    - #endif
- #ifndef directive tells the compiler to define MESSAGE only if MESSAGE is not already defined.

5.11 Files in C

- File handing in C is the process in which we create, open, read, write, and close operations on a file.
- C language provides different functions such as fopen(), fwrite(), fread(), fseek(), fprintf() to perform input, output, and many different C file operations in our program.

Need for File handling in C

1. Reusability:

Data stored in files can be accessed, modified, and deleted as needed, providing high reusability of information.

2. Portability:

- Files can be easily transferred between different systems without data loss.
- This feature minimizes the risk of coding errors and ensures seamless operation across platforms.

3. Efficiency:

- File handling in C simplifies the process of accessing and manipulating large amounts of data.
- It allows programs to efficiently retrieve specific information from files with minimal code, saving time and reducing the likelihood of errors.

4. Storage Capacity:

- Files provide a means to store vast amounts of data without the need to hold everything in memory simultaneously.

- This capability is particularly useful for handling large datasets and prevents memory overload in programs.

5.11.1 File Modes in C

- A file can be opened in one of four modes.
- The mode determines where the file is positioned when opened, and what functions are allowed.
- After you close a file, you can reopen the file in a different mode, depending on what you are doing.
- For example, you can create a file in create mode.
- The various modes in file are:
  - **Read mode**
  - **Update mode**
  - **Create mode**
  - **Append mode**

Read mode:

- Opens a file for reading of data.
- Read mode opens a file to the beginning.
- You cannot write to a file opened in read mode.
- You can reposition a binary or stream file in read mode.

Update mode:

- This mode allows both reading and writing of data.
- Update mode opens a file to the beginning.
- You can reposition a binary or stream file in update mode.
- You can update a record by repositioning the file to the beginning of the record, then writing the new data.

Create mode:

- It opens the specified file and positions it to the beginning.
- If a file by that name does not exist, the openfile() statement creates the file.
- If a file by that name exists, it overwrites the existing file, except in **VMS (Virtual Machine System).**
- VMS: 4GL (Fourth Generation Language) creates a new version of the file. It does not overwrite the file unless the file version limit is reached.
- **[Note: A fourth-generation programming language (4GL) is a high-level computer programming language that belongs to a class of languages]**
- To create a new file, open the file in create mode.
- You cannot read, position or rewind a file opened with create mode.

Append Mode:

- It allows writing data to the end of a file.
- You cannot read, position or rewind a file opened with append mode.

5.11.2 File Operations and Functions in C

1. fopen() - create a new file or open an existing file.
2. fclose() - close a file.
3. getc() - reads a character from a file.
4. putc() - writes a character to a file.
5. fscanf() - reads a set of data from a file.
6. fprintf() - writes a set of data to a file.
7. getw() - reads a integer from a file.
8. putw() - writes a integer to a file.
9. fseek() - set the position to desire point.
10. ftell() - gives current position in the file.
11. rewind() - set the position to the beginning point.

Opening a file

To open a file in C, the fopen() function is employed, specifying the filename or file path along with the desired access modes.

Syntax of fopen()

```
FILE *fopen(const char *file_name, const char *access_mode);
```

Parameters

- **file_name:** If the file resides in the same directory as the source file, provide its name; otherwise, specify the full path.

- **access_mode:** Specifies the operation for which the file is being opened.

Return Value

- If the file is successfully opened, it returns a file pointer to it.
- If the file fails to open, it returns NULL.There are many modes for opening a file:

File opening modes in C

Mode	Description	Example
R	Opens a text file in read-only mode, allowing only reading operations.	Example: fopen("demo.txt", "r")
W	When using the mode "w", fopen() initializes a text file for writing exclusively. If the file already exists, it clears its contents; otherwise, it creates a new file for writing.	Example: fopen("demo.txt", "w")
A	When employing the "a" mode, fopen() enables opening a text file in append mode. This mode permits writing data to the end of the file, preserving existing content.	Example: fopen("demo.txt", "a")
r+	When using the "r+" mode, fopen() facilitates opening a text file for both reading and writing operations. This mode grants the ability to manipulate data at any position within the file.	Example: fopen("demo.txt", "r+")
w+	This mode opens a text file for both reading and writing. If the file with same name already exists, it truncates the file to zero length; otherwise, it creates a new file for both reading and writing operations.	Example: fopen("demo.txt", "w+")

Mode	Description	Example
a+	This mode opens a text file for both reading and writing, enabling data to be appended to the end of the file without overwriting existing content.	Example: fopen("demo.txt", "a+")
Rb	Opens a binary file in read-only mode, allowing reading operations on binary data.	Example: fopen("demo.txt", "rb")
Wb	Opens a binary file in write-only mode, truncating the file to zero length if it exists or creating a new file for writing binary data.	Example: fopen("demo.txt", "wb")
ab	Opens a binary file in append mode, allowing binary data to be written to the end of the file without overwriting existing content.	Example: fopen("demo.txt", "ab")
rb+	Opens a binary file for both reading and writing operations on binary data.	Example: fopen("demo.txt", "rb+")
wb+	Opens a binary file for both reading and writing operations, truncating the file to zero length if it exists or creating a new file for reading and writing binary data.	Example: fopen("demo.txt", "wb+")
ab+	This mode opens a binary file for both reading and writing operations, allowing binary data to be appended to the end of the file without overwriting existing content.	Example: fopen("demo.txt", "ab+")

Example (Opening a file)

```
// C Program to illustrate file opening
#include <stdio.h>
#include <stdlib.h>
int main()
{
        // Declare a file pointer variable to store the value returned by fopen
        FILE* file_ptr;
        // Open the file "example.txt" in read mode
        file_ptr = fopen("example.txt", "r");

        // Check if the file is opened successfully
        if (file_ptr == NULL)
      {
printf("Failed to open the file. The program will now exit.");
                exit(1);
```

```
    }
    printf("File opened successfully!");
    return 0;
}
```

Create a File in C

- The fopen() function in C not only opens existing files but also creates a new file if it doesn't already exist.
- This behavior is achieved by using specific modes that allow file creation, such as "w", "w+", "wb", "wb+", "a", "a+", "ab", and "ab+".

```
FILE *file_ptr;
file_ptr = fopen("filename.txt", "w");
```

Example of Creating a File:

// C program to to create a file using fopen() function if it doesn't exist

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // Declare a file pointer variable to store the value returned by fopen
    FILE* file_ptr;
    // Open or create the file "filename.txt" in write mode
    file_ptr = fopen("filename.txt", "w");
    // Check if the file is opened successfully
    if (file_ptr == NULL)
   {
    printf("Failed to create/open the file. The program will now exit.");
        exit(1);
    }
    printf("File created/opened successfully!");
    return 0;
```

```
}
```

Reading from a File

- To read data from an existing file, we will use “r” mode in file opening.
- To read the file character by character, we use getc(). And to read line by line, we use fgets().

Function	Description
fscanf()	Retrieves input from a file using a formatted string and variable argument list.
fgets()	Obtains a complete line of text from the file.
fgetc()	Reads a single character from the file.
fgetw()	Reads a numerical value from the file.
fread()	Extracts a specified number of bytes from a binary file.

Example: (Read the file character by character)

```
#include <stdio.h>
#include <stdlib.h>
int main()
 {
  FILE * fp;
  char s;
  fp = fopen("file2.txt", "r");
  if (fp == NULL) {
    printf("\nCAN NOT OPEN FILE");
    exit(1);
  }
  do
  {
    s = getc(fp); // Read file character by character.
    printf("%c", s);
```

```
  }
  while (s != EOF);
  fclose(fp);
  return 0;
}
```

In the above program, the getc() function is utilized to read the file character by character until it reaches the end of the file (EOF).

Write to a File

- File writing operations in C are facilitated by functions like fprintf() and fputs().
- Additionally, C programming provides several other functions suitable for writing data to a file.

Function	Description
fprintf()	This function uses a formatted string and variable arguments list to print output to the file.
fputs()	Prints an entire line in the file along with a newline character at the end.
fputc()	Writes a single character into the file.
fputw()	Writes a number to the file.
fwrite()	Writes the specified number of bytes to the binary file.

Example:

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

```
int main()
{
    FILE *fptr;
    int num = 42;

    // Opening the file in write mode
    fptr = fopen("numbers.txt", "w");
    // Writing data to the file using different functions
    fprintf(fptr, "The number is: %d\n", num);
    fputs("This is a test line.\n", fptr);
    fputc('A', fptr);
    fputw(100, fptr);
    // Closing the file
    fclose(fptr);
    return 0;
}
```

OUTPUT:

The number is: 42
This is a test line.
A

Closing a File

- The fclose() function is utilized to close a file in C programming.
- It is essential to close a file after performing file operations in C to release system resources and ensure proper memory management.

Syntax of fclose()

```
fclose(file_pointer);
```

Here, file_pointer is a pointer to the opened file that you want to close.

Example:

```
FILE *fptr;

fptr = fopen("fileName.txt", "w");
```

// Perform file operations

fclose(fptr);

After completing the necessary file operations in C, the fclose() function is called to close the file pointed to by fptr.

Example: (File operations)

// C program to Create a file, write in it, and Close the file

```
#include <stdio.h>
#include <string.h>
int main()
{
   // Declare a pointer for the file
   FILE *diaryFile;
   // Content to be written into the file
   char diaryEntry[150] = "Diary of a Programmer - "
                   "Today, I learned about file handling in C, "
   "which feels like unlocking a new programming superpower.";

   // Opening the file "LearningDiary.txt" in write mode ("w")
   diaryFile = fopen("LearningDiary.txt", "w");
   // Verifying if the file was successfully opened
    if (diaryFile == NULL)
     {
      printf("LearningDiary.txt file could not be opened.\n");
    }
    else
    {
      printf("File opened successfully.\n");
      // Checking if there's content to write
      if (strlen(diaryEntry) > 0)
     {
       // Writing the diary entry to the file
```

```
        fputs(diaryEntry, diaryFile);
       fputs("\n", diaryFile); // New line at the end of the entry
      }
      // Closing the file to save changes
      fclose(diaryFile);
  printf("Diary entry successfully recorded in LearningDiary.txt\n");
   printf("File closed. Diary saved.\n");
   }
   return 0;
}
```

OUTPUT:

File opened successfully.
Diary entry successfully recorded in LearningDiary.txt
File closed. Diary saved.

5.11.3 Text and Binary File

Files in C can be handled either as text files or binary files:

Text Files: Text files contain human-readable characters and are typically created and edited using text editors. Functions like fscanf(), fprintf(), fgets(), and fputs() are used to read from and write to text files.

Binary Files: Binary files contain data in a format that is not human-readable. They are used for storing and retrieving structured data efficiently. Functions like fread() and fwrite() are used for binary file I/O operations

Text file

- The user can create these files easily while handling files in C.
- It stores information in the form of ASCII characters internally.
- When the file is opened, the content is readable by humans.

- It can be created by any text editor with a .txt or .rtf (rich text) extension.
- Since text files are simple, they can be edited by any text editor like Microsoft Word, Notepad, Apple Text Edit, etc.

Binary file

- It stores information in the form of 0's or 1's instead of ASCII characters.
- It is saved with the .bin extension, taking less space.
- Since it is stored in a binary number system format, it is not readable by humans.
- Binary file is more secure than a text file.

Read and Write in a Binary File

Opening a Binary File

- When intending to operate on a file in binary mode, utilize the access modes "rb", "rb+", "ab", "ab+", "wb", or "wb+" with the fopen() function.
- Additionally, employ the ".bin" file extension for binary files.

Example:

```
FILE* filePointer = fopen("example.bin", "rb");
```

In this example, the file "example.bin" is opened in binary mode for reading using the "rb" access mode.

Write to a Binary File

- When writing data to a binary file, the fwrite() function proves invaluable.
- This function allows us to store information in binary form, comprising sequences of bits (0s and 1s).

Syntax of fwrite()

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *file_pointer);
```

Parameters

ptr: A reference to the memory block holding the data intended for writing.

size: The byte size of each element to be written.

nmemb: The count of elements to be written.

file_pointer: The FILE pointer associated with the output file stream.