STRUCTURES, UNIONS AND FILE HANDLING IN C

Structure: Declaration, Definition-Array of Structures - Pointer to Structure –Nested Structures-Union: Defining union, Accessing union members. Files: File Management functions, Random access in file- Working with Text Files and Binary Files.

5.1 STRUCTURE: DECLARATION, DEFINITION

A **structure** in C is a derived or user-defined data type. We use the keyword **struct** to define a custom data type that groups together the elements of different types. The difference between an array and a structure is that an array is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.

We are often required to work with values of different data types having certain relationships among them. For example, a **book** is described by its **title** (string), **author** (string), **price** (double), **number of pages** (integer), etc. Instead of using four different variables, these values can be stored in a single **struct** variable.

a) Declare (Create) a Structure

We can create (declare) a structure by using the **"struct"** keyword followed by the structure_tag (structure name) and declare all of the members of the structure inside the curly braces along with their data types. To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member.

Syntax of Structure Declaration

The format (syntax) to declare a structure is as follows −

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

The **structure tag** is optional and each member definition is a normal variable definition, such as "int i;" or "float f;" or any other valid variable definition.

At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is **optional**.

Example

In the following example we are declaring a structure for Book to store the details of a Book −

```
struct book{
   char  title[50];
   char  author[50];
   double price;
   int   pages;
} book1;
```

Here, we declared the structure variable **book1** at the end of the structure definition. However, you can do it separately in a different statement.

b) Structure Variable Declaration

To access and manipulate the members of the structure, you need to declare its variable first. To declare a structure variable, write the structure name along with the **"struct"** keyword followed by the name of the structure variable. This structure variable will be used to access and manipulate the structure members.

Example

The following statement demonstrates how to declare (create) a structure variable

```
struct book book1;
```

Usually, a structure is declared before the first function is defined in the program, after the **include** statements. That way, the derived type can be used for declaring its variable inside any function.

c) Structure Initialization

The **initialization** of a struct variable is done by placing the value of each element inside curly brackets.

Example

The following statement demonstrates the initialization of structure

```
struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};
```

d) Accessing the Structure Members

To access the members of a structure, first, you need to declare a structure variable and then use the **dot (.) operator** along with the structure variable.

Example 1

The four elements of the struct variable book1 are accessed with the **dot (.) operator**. Hence, "book1.title" refers to the title element, "book1.author" is the author name, "book1.price" is the price, "book1.pages" is the fourth element (number of pages).

Take a look at the following example −

```c
#include <stdio.h>

struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
};

int main(){
   struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};

   printf("Title:  %s \n", book1.title);
   printf("Author: %s \n", book1.author);
   printf("Price:  %lf\n", book1.price);
   printf("Pages:  %d \n", book1.pages);
   printf("Size of book struct: %d", sizeof(struct book));
   return 0;
}
```

Output

Title: Learn C

Author: Dennis Ritchie

Price: 675.500000

Pages: 325

Size of book struct: 48

Example 2

In the above program, we will make a small modification. Here, we will put the **type definition** and the **variable declaration** together, like this −

```
struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
} book1;
```

Note that if you a declare a struct variable in this way, then you cannot initialize it with curly brackets. Instead, the elements need to be assigned individually.

```
#include <string.h>

struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
} book1;

int main(){
   strcpy(book1.title, "Learn C");
   strcpy(book1.author, "Dennis Ritchie");
   book1.price = 675.50;
   book1.pages = 325;

   printf("Title: %s \n", book1.title);
   printf("Author: %s \n", book1.author);
   printf("Price: %lf \n", book1.price);
   printf("Pages: %d \n", book1.pages);
   return 0;
}
```

Output

Title: Learn C

Author: Dennis Ritchie

Price: 675.500000

Pages: 325

5.2 ARRAY OF STRUCTURES IN C

In C programming, the **struct** keyword is used to define a derived data type. Once defined, you can declare an array of struct variables, just like an array of **int**, **float** or **char** types is declared. An array of structures has a number of use-cases such as in storing records similar to a database table where you have each row with different data types.

Usually, a struct type is defined at the beginning of the code so that its type can be used inside any of the functions. You can declare an array of structures and later on fill data in it or you can initialize it at the time of declaration itself.

Initializing a Struct Array

Let us define a struct type called **book** as follows −

```
struct book{
   char title[10];
   double price;
   int pages;
};
```

During the program, you can declare an array and initialize it by giving the values of each element inside curly brackets. Each element in the struct array is a struct value itself. Hence, we have the nested curly brackets as shown below −

```
struct book b[3] = {
   {"Learn C", 650.50, 325},
   {"C Pointers", 175, 225},
   {"C Pearls", 250, 250}
};
```

How does the compiler allocate memory for this array? Since we have an array of three elements, of **struct** whose size is 32 bytes, the array occupies "32 x 3" bytes. Each block of 32 bytes will accommodate a "title", "price" and "pages" element.

L	E	A	R	N		C				**675.50**	**325**
C	P	O	I	N	T	E	R	S		**175**	**225**
C		P	E	A	R	L	S			**250**	**250**

a) Declaring a Struct Array

You can also declare an empty struct array. Afterwards, you can either read the data in it with scanf() statements or assign value to each element as shown below −

```
struct book b[3];
strcpy(b[0].title, "Learn C");
b[0].price = 650.50;
b[0].pages=325;

strcpy(b[1].title, "C Pointers");
b[1].price = 175;
b[1].pages=225;

strcpy(b[2].title, "C Pearls");
b[2].price = 250;250
b[2].pages=325;
```

b) Reading a Struct Array

We can also accept data from the user to fill the array.

Example 1

In the following code, a **for** loop is used to accept inputs for the "title", "price" and "pages" elements of each struct element of the array.

```c
#include <stdio.h>

struct book{
   char title[10];
   double price;
   int pages;
};

int main(){

   struct book b[3];

   strcpy(b[0].title, "Learn C");
   b[0].price = 650.50;
   b[0].pages = 325;

   strcpy(b[1].title, "C Pointers");
   b[1].price = 175;
   b[1].pages = 225;
```

```c
strcpy(b[2].title, "C Pearls");
  b[2].price = 250;
  b[2].pages = 325;
  printf("\nList of Books:\n");
  for (int i = 0; i < 3; i++){
    printf("Title: %s \tPrice: %7.2lf \tPages: %d\n", b[i].title, b[i].price, b[i].pages);
  }

  return 0;
}
```

Output

List of Books:

Title: Learn C Price: 650.50 Pages: 325

Title: C Pointers Price: 175.00 Pages: 225

Title: C Pearls Price: 250.00 Pages: 325

c) <u>Sorting a Struct Array</u>

Let us take another example of struct array. Here, we will have the array of "book" struct type sorted in ascending order of the price by implementing bubble sort technique.

Note: The elements of one struct variable can be directly assigned to another struct variable by using the assignment operator.

Example

Take a look at the example

```c
#include <stdio.h>
struct book{
   char title[15];
   double price;
   int pages;
};

int main(){

   struct book b[3] = {
      {"Learn C", 650.50, 325},
      {"C Pointers", 175, 225},
      {"C Pearls", 250, 250}
   };

   int i, j;
   struct book temp;
```

```
for(i = 0; i < 2; i++){
   for(j = i; j < 3; j++){
      if (b[i].price > b[j].price){
         temp = b[i];
         b[i] = b[j];
         b[j] = temp;
      }
   }
}

printf("\nList of Books in Ascending Order of Price:\n");

for (i = 0; i < 3; i++){
   printf("Title: %s \tPrice: %7.2lf \tPages: %d\n", b[i].title, b[i].price, b[i].pages);
  }
  return 0;
}
```

Output

List of Books in Ascending Order of Price:

Title: C Pointers Price: 175.00 Pages: 225

Title: C Pearls Price: 250.00 Pages: 250

Title: Learn C Price: 650.50 Pages: 325

5.3 POINTERS TO STRUCTURES

You can define pointers to structures in the same way as you define pointers to any other variable.

a) Declaration of Pointer to a Structure

You can declare a pointer to a structure (or structure pointer) as follows −

```
struct Books *struct_pointer;
```

b) Initialization of Pointer to a Structure

You can store the address of a structure variable in the above pointer variable **struct_pointer**. To find the address of a structure variable, place the '&' operator before the structure's name as follows −

```
struct_pointer = & book1;
```

Let's store the address of a struct variable in a struct pointer variable.

```
struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
};
struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325},
struct book *strptr;
```

c) Accessing Members Using Pointer to a Structure

To access the members of a structure using a pointer to that structure, you must use the → **operator** as follows −

```
struct_pointer->title;
```

C defines the → symbol to be used with struct pointer as the **indirection operator** (also called **struct dereference operator**). It helps to access the elements of the struct variable to which the pointer reference to.

Example

In this example, **strptr** is a pointer to **struct book book1** variable. Hence, **strrptr→title** returns the title, just like **book1.title** does.

```
Open Com
```

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

struct book{
   char title[10];
   char author[20];
   double price;
   int pages;
};

int main (){
   struct book book1 = {"Learn C", "Dennis Ritchie", 675.50, 325};
   struct book *strptr;
   strptr = &book1;

   printf("Title: %s \n", strptr -> title);
   printf("Author: %s \n", strptr -> author);
   printf("Price: %lf \n", strptr -> price);
   printf("Pages: %d \n", strptr -> pages);
   return 0;
}
```

Output

Title: Learn C

Author: Dennis Ritchie

Price: 675.500000

Pages: 325

Example 2: **DECLARING A POINTER TO A STRUCT ARRAY**

We can also declare a pointer to a struct array. C uses the indirection operator (→) to access the internal elements of struct variables.

Example

The following example shows how you can declare a pointer to a struct array −

```
#include <stdio.h>

struct book {
   char title[15];
   double price;
   int pages;
};

int main(){

   struct book b[3] = {
      {"Learn C", 650.50, 325},
      {"C Pointers", 175, 225},
      {"C Pearls", 250, 250}
   };

   struct book *ptr = b;

   for(int i = 0; i < 3; i++){
printf("Title: %s \tPrice: %7.2lf \tPages: %d\n", ptr -> title, ptr -> price, ptr -> pages);
    ptr++;
  }
 return 0;
}
```

Output

Title: Learn C Price: 650.50 Pages: 325

Title: C Pointers Price: 175.00 Pages: 225

Title: C Pearls Price: 250.00 Pages: 250

5.4 NESTED STRUCTURES

C language allows us to insert one structure into another as a member. This process is called nesting and such structures are called nested structures. There are two ways in which we can nest one structure into another:

a). Embedded Structure Nesting

In this method, the structure being nested is also declared inside the parent structure.

Example

```
struct parent {
   int member1;
   struct member_str member2 {
      int member_str1;
      char member_str2;
      ...
   }
   ...
}
```

b). Separate Structure Nesting

In this method, two structures are declared separately and then the member structure is nested inside the parent structure.

Example

```
struct member_str {
   int member_str1;
   char member_str2;
   ...
}
struct parent {
   int member1;
   struct member_str member2;
   ...
}
```

One thing to note here is that the declaration of the structure should always be present before its definition as a structure member. For example, the **declaration below is invalid** as the struct mem is not defined when it is declared inside the parent structure.

```
struct parent {
   struct mem a;
};

struct mem {
```

```
    int var;
};
```

c) Accessing Nested Members

We can access nested Members by using the same (.) dot operator two times as shown:

```
str_parent.str_child.member;
```

Example of Structure Nesting

// C Program to illustrate structure nesting along with forward declaration

```
#include <stdio.h>
// child structure declaration
struct child {
    int x;
    char c;
};
// parent structure declaration
struct parent {
    int a;
    struct child b;
}
// driver code
int main()
{
    struct parent var1 = { 25, 195, 'A' };
    // accessing and printing nested members
    printf("var1.a = %d\n", var1.a);
    printf("var1.b.x = %d\n", var1.b.x);
    printf("var1.b.c = %c", var1.b.c);
    return 0;
}
```

Output

```
var1.a = 25
```

```
var1.b.x = 195

var1.b.c = A
```

5.5 UNIONS IN C

A **union** is a special data type available in C that allows 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 multiple purpose.

All the members of a union share the same memory location. Therefore, if we need to use the same memory location for two or more members, then union is the best data type for that. The largest union member defines the size of the union.

a) Defining a Union

Union variables are created in same manner as structure variables. The keyword **union** is used to define unions in C language.

Syntax

Here is the syntax to define a **union** in C language −

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

The "union tag" is optional and each member definition is a normal variable definition, such as "int i;" or "float f;" or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables.

b) Accessing the Union Members

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type.

Syntax

Here is the syntax to access the members of a union in C language –

```
union_name.member_name;
```

c) Initialization of Union Members

You can initialize the members of the union by assigning the value to them using the assignment (=) operator.

Syntax

Here is the syntax to initialize members of union −

```
union_variable.member_name = value;
```

Example

The following code statement shows to initialization of the member "i" of union "data" − `data.i = 10;`

d) Examples of Union

Example 1

The following example shows how to use unions in a program −

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

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

int main(){
   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy(data.str, "C Programming");

   printf("data.i: %d \n", data.i);
   printf("data.f: %f \n", data.f);
   printf("data.str: %s \n", data.str);
   return 0;
}
```

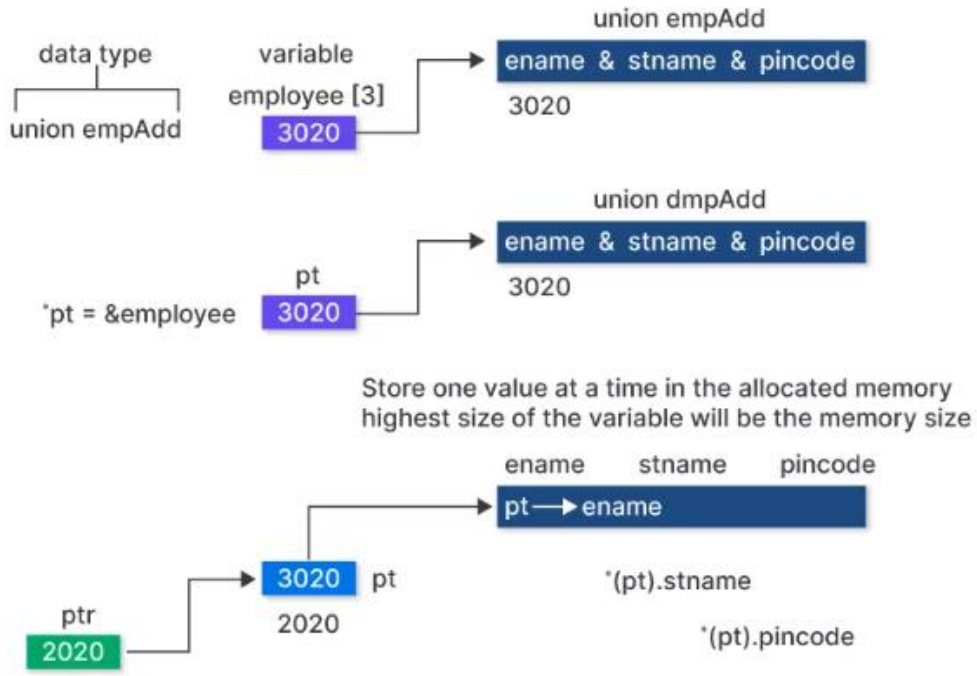
Output

When the above code is compiled and executed, it produces the following result −

data.i: 1917853763
data.f: 4122360580327794860452759994368.000000
data.str: C Programming

Here, we can see that the values of **i** and **f** (members of the union) show **garbage values** because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Accessing union Members in C



When you define a union, you're essentially creating a container for multiple members, each of which can have a different data type. These members share the same memory space that you can use to store any of the union's members interchangeably. But you can actively access only one of the members. There are two methods of accessing members of a union in C-

1. Using the dot operator (.)
2. Using the arrow operator/ 'this' pointer (->)

However, note that accessing members of a union in C programs involves working with a specific/ distinct type of data. The most commonly used member types are:

1. Integer Members: These members store whole numbers (integer type).
2. Floating-Point Members: These members store decimal numbers (floats or doubles).
3. Character Members: These members store individual characters.
4. Array Members: These members store collections of elements of the same data type.

5. Pointer Members: These members store memory addresses, pointing to specific locations in memory.

We can use the dot operator to access most normal data types of union member's values, including integer, double, character, string (character array) and floating-point values. But when working with pointer type members/ union variables, we must use the arrow pointer method. Let's look at both methods!

Accessing Normal Members of Union in C Using Dot Operator

Accessing normal members of a union in C using the dot operator (.) is straightforward and is used when the union variable itself is not a pointer. All you have to do is use the dot operator (.) followed by the member name to access and manipulate the members of the union.

Syntax:

VariableName.member_name;

Here, we specify the name of the union variable (variableName) and the name of the union member (member_name), connected with a dot operator. We have already seen how to use this approach in the example above, but here is another sample C program demonstrating how to access members of a union in C using the dot operator:

5.6 FILE HANDLING IN C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

5.6.1 Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position

11	rewind()	sets the file pointer to the beginning of the file

---

1) **Opening File: fopen()**

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

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

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like **"c://some_folder/some_file.ext"**.
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

ode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode

a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
#include<stdio.h>
void main( )
{
FILE *fp ;
char ch ;
fp = fopen("file_handle.c","r") ;
while ( 1 )
{
ch = fgetc ( fp ) ;
if ( ch == EOF )
break ;
printf("%c",ch) ;
```

```
}
close (fp ) ;
```

Output

The content of the file will be printed.

```
#include;
void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

2) **Closing File: fclose()**

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

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

3) Writing File : fprintf() function

The fprintf() function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
      fprintf(fp, "Hello file by fprintf...\n");//writing data into file
      fclose(fp);//closing file
   }
```

4) Reading File: fscanf() function

The fscanf() function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

```
    #include <stdio.h>
main(){
  FILE *fp;
  char buff[255];         //creating char array to store data of file
      fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF)
{
 printf("%s ", buff );
 }
      fclose(fp);
```

```
}
```

Output:

```
Hello file by fprintf...
```

C File Example: Storing employee information

Let's see a file handling example to store employee information as entered by user from console. We are going to store id, name and salary of the employee.

```
#include <stdio.h>
void main()
{
  FILE *fptr;
      int id;
      char name[30];
      float salary;
  fptr = fopen("emp.txt", "w+");        /*  open for writing */
  if (fptr == NULL)
      {
         printf("File does not exists \n");
    return;
  }
  printf("Enter the id\n");
      scanf("%d", &id);
      fprintf(fptr, "Id= %d\n", id);
  printf("Enter the name \n");
      scanf("%s", name);
      fprintf(fptr, "Name= %s\n", name);
      printf("Enter the salary\n");
  scanf("%f", &salary);
      fprintf(fptr, "Salary= %.2f\n", salary);
```

```
    fclose(fptr);
     }
```

Output:

Enter the id
1
Enter the name
Sonoo
Enter the salary
120000

C fputc() and fgetc() Functions

Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

```
int fputc(int c, FILE *stream)
```

Example

```
     #include <stdio.h>
     main()
     {     FILE *fp;
   fp = fopen("file1.txt", "w");           //opening file
 fputc('a',fp);                          //writing single character into file
 fclose(fp);                              //closing file
  }
```

file1.txt

a

Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

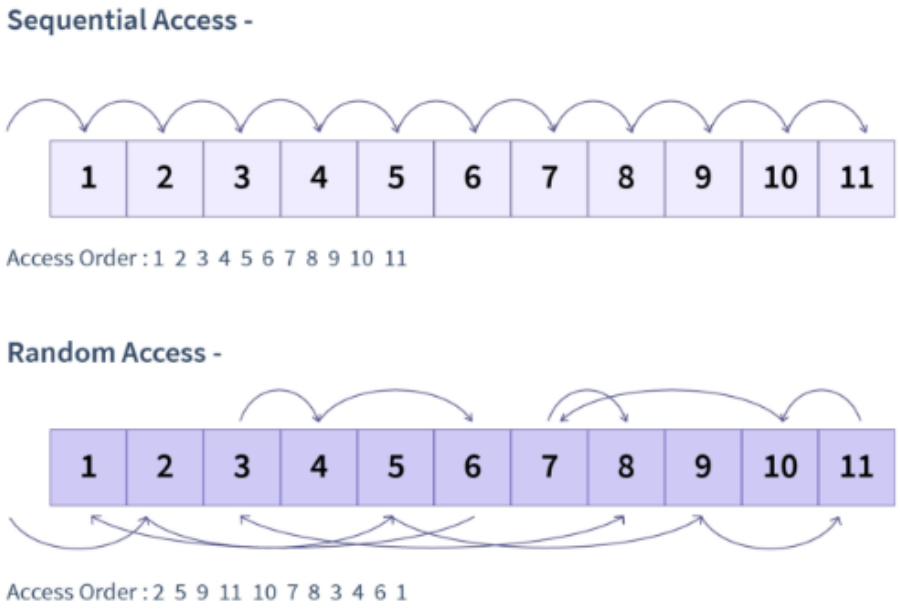
```
int fgetc(FILE *stream)
#include<stdio.h>
    #include<conio.h>
    void main(){
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("myfile.txt","r");

while((c=fgetc(fp))!=EOF){
    printf("%c",c);
    }
    fclose(fp);
getch();
    }
```

myfile.txt

this is simple text message

5.7 RANDOM ACCESSING FILES IN C LANGUAGE

Random file access in C, enabling direct data reading or writing without processing all preceding data. Distinct from sequential access, it offers enhanced flexibility for data manipulation. Random access, ideal for large files, involves functions like ftell(), fseek(), and rewind(). This method, akin to choosing a song on a CD, requires more coding but offers superior efficiency and flexibility in file handling.



How to use the ftell() function in C

Highlights:

1. ftell() is used to find the position of the file pointer from the starting of the file.
2. Its syntax is as follows:

```
ftell(FILE *fp)
```

In C, the function ftell() is used to determine the file pointer's location relative to the file's beginning. ftell() has the following syntax:

```
pos = ftell(FILE *fp);
```

Where fp is a file pointer and pos holds the current position i.e., total bytes read (or written). For Example: If a file has 20 bytes of data and if the ftell() function returns 5 it means that 5 bytes have already been read (or written). Consider the below program to understand the ftell() function:

First, let us consider a file - Scaler.txt which contains the following data:

Scaler is amazing

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    fp=fopen("scaler.txt","r");
    if(!fp)
    {
        printf("Error: File cannot be opened\n") ;
        return 0;
    }
```

```c
    //Since the file pointer points to the starting of the file, ftell
    printf("Position pointer in the beginning : %ld\n",ftell(fp));

    char ch;
    while(fread(&ch,sizeof(ch),1,fp)==1)
    {
        //Here, we traverse the entire file and print its contents unt
        printf("%c",ch);
    }

    printf("\nSize of file in bytes is : %ld\n",ftell(fp));
    fclose(fp);
    return 0;
}
```

Output:

```
Position pointer in the beginning : 0
Scaler is amazing
Size of file in bytes is : 17
```

We can observe that in the beginning, ftell returns 0 as the pointer points to the beginning and after traversing completely we print each character of the file till the end, and now ftell returns 17 as it is the size of the file.

How to use the rewind() function in C

Highlights:

1. rewind() is used to move the file pointer to the beginning of the file.
2. Its syntax is as follows:

```
rewind(FILE *fp);
```

The file pointer is moved to the beginning of the file using this function. It comes in handy when we need to update a file. The following is the syntax:

```
rewind(FILE *fp);
```

Here, fp is a file pointer of type FILE. Consider the following program to understand the rewind() function:

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    fp = fopen("scaler.txt","r");
    if(!fp)
    {
        printf("Error in opening file\n");
        return 0;
    }
    //Initially, the file pointer points to the starting of the file.
    printf("Position of the pointer : %ld\n",ftell(fp));

    char ch;
    while(fread(&ch,sizeof(ch),1,fp)==1)
    {
        //Here, we traverse the entire file and print its contents unt
        printf("%c",ch);
    }
    printf("Position of the pointer : %ld\n",ftell(fp));

    //Below, rewind() will bring it back to its original position.
    rewind(fp);
    printf("Position of the pointer : %ld\n",ftell(fp));

    fclose(fp);
    return 0;
}
```

Output:

```
Position of the pointer : 0
Scaler is amazing
```

```
Position of the pointer : 17
Position of the pointer : 0
```

We can observe that firstly when ftell is called, it returns 0 as the position of the pointer is at the beginning, and then after traversing the file, when ftell is called, 17 is returned, which is the size of the file. Now when rewind(fp) is called, the pointer will move to its original position, which is 0. So last ftell returns 0.

How to use the fseek() function in C

Highlights:

1. The fseek() function moves the file position to the desired location.
2. Its syntax is:

```
int fseek(FILE *fp, long displacement, int origin);
```

To shift the file position to a specified place, use the fseek() function.

Syntax:

```
int fseek(FILE *fp, long displacement, int origin);
```

The various components are as follows:

- *fp* – file pointer.
- *displacement* - represents the number of bytes skipped backwards or forwards from the third argument's location. It's a long integer that can be either positive or negative.
- *origin* – It's the location relative to the displacement. It accepts one of the three values listed below.

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

Here is the list of common operations that we can perform using the fseek() function.

Here is the list of common operations that we can perform using the fseek() function.

Operation	Description
fseek(fp, 0, 0)	This takes us to the beginning of the file.
fseek(fp, 0, 2)	This takes us to the end of the file.
fseek(fp, N, 0)	This takes us to (N + 1)th bytes in the file.
fseek(fp, N, 1)	This takes us N bytes forward from the current position in the file.

fseek(fp, -N, 1)	This takes us N bytes backward from the current position in the file.
fseek(fp, -N, 2)	This takes us N bytes backward from the end position in the file.

Let us see the below program to understand the fseek() function:

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    fp = fopen("scaler.txt","r");
    if(!fp)
    {
        printf("Error: File cannot be opened\n");
        return 0;
    }
    //Move forward 6 bytes, thus we won't be seeing the first 6 bytes
    fseek(fp, 6, 0);
    char ch;
    while(fread(&ch,sizeof(ch),1,fp)==1)
    {
        //Here, we traverse the entire file and print its contents unt
        printf("%c",ch);
    }

    fclose(fp);
    return 0;
}
```

Output:

```
is amazing
```

We can observe that when fseek(fp,6,0) the pointer moves to the 7th byte in the file, or we can say 6 bytes forward from the beginning, So when we traverse the file from that position, we receive output as is amazing.

File Mode Combinations

Highlights:

File Mode combinations allow us to accomplish reading and writing operations simultaneously.

In general, you can only read from or write to a text file, not simultaneously. A binary file allows you to read and write to the same file. What you can accomplish with each combination is shown in the table below:

Combination	Type of File	Operation
r	text	read
rb+	binary	read
r+	text	read, write
r+b	binary	read, write
rb+	binary	read, write
w	text	write, create, truncate
wb	binary	write, create, truncate
w+	text	read, write, create, truncate
w+b	binary	read, write, create, truncate
wb+	binary	read, write, create, truncate
a	text	write, create
ab	binary	write, create
a+	text	read, write, create
a+b	binary	write, create
ab+	binary	write, create

Creating a Random-Access File

Functions like fopen() can be used to create files if they do not exist.

Functions like fopen() can be used to create files if they do not exist. This can be seen in the example below:

```
#include<stdio.>
int main()
{
   char ch;
      // file pointer
   FILE *fp;
    // open and creates file in write mode if it does not exist.
   fp = fopen("char", "w");
   if (fptr != NULL)
   {
      printf("File created successfully!\n");
   }
   else
   {
      printf("Failed to create the file.\n");
      return 0;
   }
   fclose(fp)
   return 0;
}
```

Writing Data Randomly to a Random-Access File

The program writes data to the file "student.txt". It stores data at precise points in the file using a mix of fseek() and fwrite(). The file position pointer is set to a given place in the file by fseek(), and then the data is written by fwrite(). Let us see the code below:

```
#include <stdio.h>

// Student structure definition
struct Student {
    char name[20]; // student name
    int roll_number; // roll number
};
int main()
{
    FILE *fp; // file pointer
        // The below line creates a student object with default values
    struct Student s = {"", 0};
        // fopen opens the file, and exits if file cannot be opened
    if (!(fp = fopen( "student.txt", "r+" )))
    {
      printf("File cannot be opened.");
      return 0;
    }
       // The user will enter information which will be copied to the file
    while(1)
    {
        // require the user to specify roll number
        printf("Enter roll number from (1 to 100) , -1 to end input : ");
        scanf("%d",&s.roll_number);
        if(s.roll_number == -1)
            break;
                // require the user to specify name
        printf("Enter name : ");
        scanf("%s",s.name);
                fseek(fp,(s.roll_number-1)*sizeof(s),0);
        fwrite(&s, sizeof(s), 1, fp);
    }
```

```
    fclose(fp); // fclose closes the file
    return 0;
}
```

Output:

```
Enter roll number from (1 to 100) , -1 to end input : 1
Enter name : Scaler
Enter roll number from (1 to 100) , -1 to end input : 10
Enter name : Aaradhya
Enter roll number from (1 to 100) , -1 to end input : -1
```

5.8 WORKING WITH TEXT FILES AND BINARY FILES.

Files is collection of records (or) it is a place on hard disk, where data is stored permanently.

Types of Files:

There are two types of files in C language which are as follows −

- Text file
- Binary File

Text File

- It contains alphabets and numbers which are easily understood by human beings.
- An error in a text file can be eliminated when seen.
- In text file, the text and characters will store one char per byte.
- For example, the integer value 4567 will occupy 2 bytes in memory, but, it will occupy 5 bytes in text file.
- The data format is usually line-oriented. Here, each line is a separate command.

Binary file

- It contains 1's and 0's, which are easily understood by computers.
- The error in a binary file corrupts the file and is not easy to detect.

- In binary file, the integer value 1245 will occupy 2 bytes in memory and in file.
- A binary file always needs a matching software to read or write it.
- For example, an MP3 file can be produced by a sound recorder or audio editor, and it can be played in a music player.
- MP3 file will not play in an image viewer or a database software.

Files are classified into following

- **Sequential files** − Here, data is stored and retained in a sequential manner.
- **Random access Files** − Here, data is stored and retrieved in a random wa

Differentiators between these two file types:

Aspect	Binary File	Text File
Data Representation	Stores data in binary format (***1s*** and ***0s***).	Stores data as ASCII characters, making it human-readable.
Use Cases	It is ideal for storing custom data like images, audio, and mixed data types.	It is suited for storing user-friendly, plain text data. Commonly used for documents, configuration files, etc.
Memory Consumption	Occupies memory based on the number of bytes in binary format.	Uses more memory due to character-based storage (1 byte per character).
Newline Handling	No automatic conversion of newline characters.	Converts newline characters to carriage return-line feed combinations.
Accessibility	Requires custom applications or software for data interpretation.	Can be viewed and edited using simple text editors.

End of File Marker	Typically tracks the end of the file based on the number of characters present.	Uses a unique ***ASCII value (26)*** as an end-of-file marker.
Data Security	Data is encrypted, making it secure but challenging to understand.	Data is less secure, but errors can be easily identified and corrected.
Error Handling	A single error can corrupt the entire file, challenging to rectify.	Errors are easier to spot and fix due to human-readable format.

C File Operations

C file operations refer to the different possible operations that we can perform on a file in C such as:

1. Creating a new file – **fopen() with attributes as "a" or "a+" or "w" or "w+"**
2. Opening an existing file – **fopen()**
3. Reading from file – **fscanf() or fgets()**
4. Writing to a file – **fprintf() or fputs()**
5. Moving to a specific location in a file – **fseek(), rewind()**
6. Closing a file – **fclose()**

Functions for C File Operations

File operation	Declaration & Description
fopen() - To open a file	Declaration: FILE *fopen (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=fopen ("filename", "'mode"); Where, fp – file pointer to the data type "FILE". filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.
fclose() - To close a file	Declaration: int fclose(FILE *fp); fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below. fclose (fp);
fgets() - To read a file	Declaration: char *fgets(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. fgets (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer
fprintf() - To write into a file	Declaration: int fprintf(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);

Read and Write in a Binary File

Till now, we have only discussed text file operations. The operations on a binary file are similar to text file operations with little difference.

Opening a Binary File

To open a file in binary mode, we use the rb, rb+, ab, ab+, wb, and wb+ access mode in the fopen() function. We also use the .bin file extension in the binary filename.

Example

```
fptr = fopen("filename.bin", "rb");
```

Write to a Binary File

We use fwrite() function to write data to a binary file. The data is written to the binary file in the from of bits (0's and 1's).

Syntax of fwrite()

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

Parameters:

- **ptr:** pointer to the block of memory to be written.
- **size:** size of each element to be written (in bytes).
- **nmemb:** number of elements.
- **file_pointer**: FILE pointer to the output file stream.

Return Value:

- Number of objects written.

Example:

Program to write to a Binary file using fwrite() C

```
// C program to write to a Binary file using fwrite()
#include <stdio.h>
#include <stdlib.h>
struct threeNum {
   int n1, n2, n3;
};
int main()
{
   int n;
   // Structure variable declared here.
   struct threeNum num;
   FILE* fptr;
   if ((fptr = fopen("C:\\program.bin", "wb")) == NULL) {
      printf("Error! opening file");
      // If file pointer will return NULL
      // Program will exit.
      exit(1);
```

```
    }
    int flag = 0;
    // else it will return a pointer to the file.
    for (n = 1; n < 5; ++n) {
        num.n1 = n;
        num.n2 = 5 * n;
        num.n3 = 5 * n + 1;
        flag = fwrite(&num, sizeof(struct threeNum), 1,
                      fptr);
    }
    // checking if the data is written
    if (!flag) {
        printf("Write Operation Failure");
    }
    else {
        printf("Write Operation Successful");
    }
    fclose(fptr);
     return 0;
}
```

Output

Write Operation Successful