Unit V: Structures and File System

Structures

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

- if you want to access structure members in c, structure variable should be declared.
- many structure variables can be declared for same structure and memory will be allocated for each separately.
- it is a best practice to initialize a structure to null while declaring, if we don't assign any values to structure members.

```
struct struct_name {
   datatype member1_name;
   datatype member2_name;
   datatype member3_name;
   …
};
```

struct_name can be anything of your choice.
Members data type can be same or different.
Once we have declared the structure we can use the struct name as a data type like int, float etc.

Declare Structure Variable

```
Struct  struct_name  var_name;
Or
Struct struct_name {
   datatype member1_name;
   datatype member2_name;
   datatype member3_name;
   …
} var_name;
```

Accessing data members of a structure using a struct variable

```
Var_name.member1_name;
Var_name.member2_name;
…
```

There are three ways to do this.
1) Using Dot(.) operator
var_name.memeber_name = value;
2) All members assigned in one statement
struct struct_name var_name =
{value for memeber1, value for memeber2 …so on for all the members}

Uses of Structures In C:

C Structures can be used to store huge data. Structures act as a database.
C Structures can be used to send data to the printer.
C Structures can interact with keyboard and mouse to store the data.
C Structures can be used in drawing and floppy formatting.
C Structures can be used to clear output screen contents.

C Structures can be used to check computer's memory size etc.

Difference Between C Variable, C Array And C Structure:

- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types and Data types can be int, char, float, double and long double etc.

Using normal variable	**Using pointer variable**

Syntax:
struct tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	struct tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
struct student
{
int mark;
char name[10];
float average;
};
	Example:
	struct student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring structure using normal variable: struct student report;	Declaring structure using pointer variable: struct student *report, rep;

Initializing structure using normal variable:
struct student report = {100, "Mani", 99.5};
	Initializing structure using pointer variable:
	struct student rep = {100, "Mani", 99.5};
	report = &rep;

Accessing structure members using normal variable:
report.mark;
report.name;
report.average;
	Accessing structure members using pointer variable:
	report -> mark;
	report -> name;
	report -> average;

Example: This program is used to store and access "id, name and percentage" for one student.

```
#include <stdio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
int main()
{
        struct student record = {0}; //Initializing to null
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
        return 0;
}
```

OUTPUT:

```
Id                                      is:                                     1
Name                                    is:                                  Raju
Percentage is: 86.500000
```

Nested Structure in C: Struct inside another struct

We can use a structure inside another structure. The struct struct_name acts as a new data type and you can include it in another struct.

Example: Structure 1: stu_address

```
struct stu_address
{
     int street;
     char *state;
     char *city;
     char *country;
}
```

Structure 2: stu_data

```
struct stu_data
{
    int stu_id;
    int stu_age;
    char *stu_name;
    struct stu_address stuAddress;
}
```

Use of typedef in Structure

typedef makes the code short and improves readability. It is like an alias of struct.

```
struct home_address {
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
};
...
struct home_address var;
```

```
var.town = "Agra";
```

Code using tyepdef

```
typedef struct home_address{
  int local_street;
  char *town;
  char *my_city;
  char *my_country;
}addr;
..
..
addr var1;
var.town = "Agra";
```

C – Array of Structures

C structure is collection of different datatypes (variables) which are grouped together. whereas, array of structures is a collection of structures. This is also called as structure array in c.

Example:

This program is used to store and access “id, name and percentage” for 3 students.

```
#include <stdio.h>
#include <string.h>
struct student
{
     int id;
     char name[30];
     float percentage;
};
int main()
{
     int i;
     struct student record[2];
     // 1st student's record
     record[0].id=1;
     strcpy(record[0].name, "raju");
     record[0].percentage = 86.5;
     // 2nd student's record
     record[1].id=2;
     strcpy(record[1].name, "surendren");
     record[1].percentage = 90.5;
     // 3rd student's record
     record[2].id=3;
     strcpy(record[2].name, "thiyagu");
     record[2].percentage = 81.5;
     for(i=0; i<3; i++)
     {
         printf("     records of student : %d \n", i+1);
         printf(" id is: %d \n", record[i].id);
         printf(" name is: %s \n", record[i].name);
```

```
        printf(" percentage is: %f\n\n",record[i].percentage);
    }
    return 0;
}
```

Output:

```
records of student : 1
id is: 1
name is: raju
percentage is: 86.500000

records of student : 2
id is: 2
name is: surendren
percentage is: 90.500000

records of student : 3
id is: 3
name is: thiyagu
percentage is: 81.500000
```

C Pointers to struct

```
struct name {
member1;
member2;
.
.
};

int main()
{
struct name *ptr, Harry;
}
```

A pointer ptr of type >struct name> is created. That is, ptr is a pointer to struct.
Example: Access members using Pointer
To access members of a structure using pointers, we use the -> operator.

```
#include <stdio.h>
struct person
{
int age;
float weight;
};
int main()
{
struct person *personPtr, person1;
personPtr = &person1;
printf("Enter age: ");
scanf("%d", &personPtr->age);
```

```
 printf("Enter weight: ");
scanf("%f", &personPtr->weight);
printf("Displaying:\n");
printf("Age: %d\n", personPtr->age);
printf("weight: %f", personPtr->weight);
return 0;
}
```

By the way,
personPtr->age is equivalent to (*personPtr).age
personPtr->weight is equivalent to (*personPtr).weight

Dynamic memory allocation of structs
Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time.

Example: Dynamic memory allocation of structs

```
#include <stdio.h>
#include <stdlib.h>
struct person {
int age;
float weight;
char name[30];
};
int main()
{
struct person *ptr;
int i, n;
printf("Enter the number of persons: ");
scanf("%d", &n);
 // allocating memory for n numbers of struct person
ptr = (struct person*) malloc(n * sizeof(struct person));
for(i = 0; i < n; ++i)
{
 printf("Enter first name and age respectively: ");
// To access members of 1st struct person,
// ptr->name and ptr->age is used
// To access members of 2nd struct person,
// (ptr+1)->name and (ptr+1)->age is used
scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
  }
 printf("Displaying Information:\n");
 for(i = 0; i < n; ++i)
 printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);
return 0;
}
```

Output:

```
Enter the number of persons:  2
Enter first name and age respectively:  Harry 24
Enter first name and age respectively:  Gary 32
Displaying Information:
Name: Harry   Age: 24
Name: Gary    Age: 32
```

In the above example, n number of struct variables are created where n is entered by the user. To allocate the memory for n number of struct person, we used,
ptr = (struct person*) malloc(n * sizeof(struct person));

Then, we used the ptr pointer to access elements of person.
A structure can be passed to any function from main function or from any sub function.
Structure definition will be available within the function only.
It won't be available to other functions unless it is passed to those functions by value or by address(reference).
Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

Passing Structure To Function In C:
It can be done in below 3 ways.
Passing structure to a function by value
Passing structure to a function by address(reference)
No need to pass a structure – Declare structure variable as global

Example program – passing structure to function in c by value:
In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

```
#include <stdio.h>
#include <string.h>
 struct student
{
            int id;
            char name[20];
            float percentage;
};
 void func(struct student record);
 int main()
{
            struct student record;
            record.id=1;
            strcpy(record.name, "Raju");
            record.percentage = 86.5;
            func(record);
            return 0;
}
void func(struct student record)
```

```
 {
            printf(" Id is: %d \n", record.id);
            printf(" Name is: %s \n", record.name);
            printf(" Percentage is: %f \n", record.percentage);
 }
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

In this program, the whole structure is passed to another function by address. It means only the address of the structure is passed to another function. The whole structure is not passed to another function with all members and their values. So, this structure can be accessed from called function by its address.

```
#include <stdio.h>
#include <string.h>
struct student
{
           int id;
           char name[20];
           float percentage;
};
void func(struct student *record);
int main()
{
          struct student record;
          record.id=1;
          strcpy(record.name, "Raju");
          record.percentage = 86.5;
         func(&record);
          return 0;
}
void func(struct student *record)
{
          printf(" Id is: %d \n", record->id);
          printf(" Name is: %s \n", record->name);
          printf(" Percentage is: %f \n", record->percentage);
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

Structure variables also can be declared as global variables as we declare other variables in C. So, When a structure variable is declared as global, then it is visible to all the functions in a program. In this scenario, we don’t need to pass the structure to any function separately.

```
#include <stdio.h>
#include <string.h>
struct student
{
        int id;
        char name[20];
        float percentage;
};
struct student record; // Global declaration of structure
void structure_demo();
int main()
{
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
        structure_demo();
        return 0;
}

void structure_demo()
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

OUTPUT:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

Unions

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.

Defining a Union

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

Syntax:

```
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 but it is optional. Here is the way you would define a union type named Data having three members i, f, and str

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

The memory occupied by a union will be large enough to hold the largest member of the union. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {
   union Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
   return 0;
}
```

Output:

```
Memory size occupied by data : 20
```

Accessing 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.

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

Bit-fields

The declaration of a bit-field has the following form inside a structure −

```
struct {
   type [member_name] : width ;
};
```

type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows −

```
struct {
   unsigned int age : 3;
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value.

Example:

```
#include <stdio.h>
#include <string.h>
struct {
   unsigned int age : 3;
} Age;
```

```
int main( ) {
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );
   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );
   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );
   return 0;
}
```

Output:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Basics of Formatted Input/Output in C

- I/O is essentially done one character (or byte) at a time
- stream -- a sequence of characters flowing from one place to another
    - input stream: data flows from input device (keyboard, file, etc) into memory
    - output stream: data flows from memory to output device (monitor, file, printer, etc)
- Standard I/O streams (with built-in meaning)
    - stdin: standard input stream (default is keyboard)
    - stdout: standard output stream (defaults to monitor)
    - stderr: standard error stream
    - stdio.h -- contains basic I/O functions
    - scanf: reads from standard input (stdin)
    - printf: writes to standard output (stdout)
- Formatted I/O -- refers to the conversion of data to and from a stream of characters, for printing (or reading) in plain text format
- The basic format of a printf function call is:
  ```
  printf (format_string, list_of_expressions);
  ```
  where:
  format_string is the layout of what's being printed
  list_of_expressions is a comma-separated list of variables or expressions yielding results to be inserted into the output

- To output string literals, just use one parameter on printf, the string itself
  ```
  printf("Hello, world!\n");
  printf("Greetings, Earthling\n\n");
  ```
  Variable length argument list
  In programming, there happens situation when you want your function to accept variable number of arguments.
  Syntax:
  ```
  return_type function_name(parameter_list, int num, ...);
  ```

- **Return type** - Specify function return type.
- **Function name** - Name of the function.
- **Parameter list** - This is optional for var-args function. If your function accepts parameter other than var-args, then pass it before var-args.
- **int num** - int num specify number of variable length arguments passed.
- **Ellipsis symbol ...** - Specify that the function is ready to accept n number of arguments.

Using variable length arguments in a function
Follow six simple steps to implement variable length arguments in your function.

1. Include stdarg.h header file to access variable length argument.
2. Define a function to accept variable length arguments.
   void myfunction(int num, ...);
3. Inside function create a va_list type variable.
   va_list list;
4. Initialize list variable using va_start macro. va_start macro accepts two parameters. First va_list type variable and number of arguments in the list.
   va_start(list, num);
5. Run a loop from 1 to number of arguments. Inside the loop, use va_arg to get next argument passed as variable length arguments. va_arg accepts two parameter. First va_list type variable from where to fetch values. Second, type (data type) of value you want to retrieve.
   va_arg(list, int);
6. Finally release memory occupied by va_list using va_end.
   va_end(list);

Example:

```
/* C program to find maximum among n arguments  */

#include <stdio.h>
#include <stdarg.h> // Used for var-args
#include <limits.h> // Used for INT_MIN
/* Variable length arguments function declaration */
int maximum(int num, ...);
int main()
{
   /*  Test var-args with some sample values.     */
   printf("Max(10,2) = %d\n", maximum(2, 10, 2));
   printf("Max(4,2,3,0) = %d\n", maximum(4, 4, 2, 3, 0));

   printf("Max(1,2,3,4,10,20,30) = %d\n", maximum(7, 1, 2, 3, 4, 10, 20, 30));
   printf("Max(100,10,0,1,2,3,4,10,20,12) = %d\n", maximum(10, 100, 10, 0, 1, 2, 3, 4, 10, 20,
12));
   return 0;
}

/* Variable length arguments function definition. num  Total number of arguments passed.
 * ...  Variable length arguments Returns maximum value among n integer  */
int maximum(int num, ...)
```

```
{
    int max = INT_MIN;
    int count; // Loop counter variable
    int value; // Store the value of current argument in var-args

    // Declare va_list type variable
    va_list list;

    // Initialize the list
    va_start(list, num);

    /* Run loop from 1 to number of arguments passed    */
    for(count=1; count<=num; count++)
    {
        // Get next argument in list
        value = va_arg(list, int);

        /* If current argument is greater than max  then store it in max.     */
        if(value > max)
            max = value;
    }

    // Clean the list
    va_end(list);

    // Finally return max argument in list
    return max;
}
```

Output -

```
Max(10,2) = 10
Max(4,2,3,0) = 4
Max(1,2,3,4,10,20,30) = 30
Max(100,10,0,1,2,3,4,10,20,12) = 100
```

File Handling

A file is a container in computer storage devices used for storing data.

Need of Files

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

Types of Files

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

1. Text files
2. Binary files

1. Text files
Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad. When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents. They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files
Binary files are mostly the .bin files in your computer. Instead of storing data in plain text, they store it in the binary form (0's and 1's).They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations
In C, you can perform four major operations on files, either text or binary:

- Creating a new file
- Opening an existing file
- Closing a file
- Reading from and writing information to a file

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

```
FILE *fptr;
```
Opening a file - for creation and edit
Opening a file is performed using the fopen() function defined in the stdio.h header file.

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

```
ptr = fopen("fileopen","mode")
```
For Example,

```
fopen("E:\\cprogram\\newprogram.txt","w");
fopen("E:\\cprogram\\oldprogram.bin","rb");
```

Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.
The writing mode allows you to create and edit (overwrite) the contents of the file.
Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode 'rb'.
The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.

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

Closing a File

The file (both text and binary) should be closed after reading/writing. Closing a file is performed using the fclose() function.

```
fclose(fptr);
```

Here, fptr is a file pointer associated with the file to be closed.

Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf(). They are just the file versions of printf() and scanf(). The only difference is that fprint() and fscanf() expects a pointer to the structure FILE.

Example 1: Write to a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int num;
  FILE *fptr;
  // use appropriate location if you are using MacOS or Linux
  fptr = fopen("C:\\program.txt","w");
  if(fptr == NULL)
  {
    printf("Error!");
    exit(1);
  }
  printf("Enter num: ");
  scanf("%d",&num);
```

```
   fprintf(fptr,"%d",num);
   fclose(fptr);
   return 0;
}
```

This program takes a number from the user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   fscanf(fptr,"%d", &num);
   printf("Value of n=%d", num);
   fclose(fptr);

   return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from Example 1, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in a similar way.

Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the fwrite() function. The functions take four arguments:

- address of data to be written in the disk
- size of data to be written in the disk
- number of such type of data
- pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }
  for(n = 1; n < 5; ++n)
  {
    num.n1 = n;
    num.n2 = 5*n;
    num.n3 = 5*n + 1;
    fwrite(&num, sizeof(struct threeNum), 1, fptr);
  }
  fclose(fptr);

  return 0;
}
```

In this program, we create a new file program.bin in the C drive. We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.
Now, inside the for loop, we store the value into the file using fwrite().The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.
Since we're only inserting one instance of num, the third parameter is 1. And, the last parameter *fptr points to the file we're storing the data.
Finally, we close the file.

Reading from a binary file

Function fread() also take 4 arguments similar to the fwrite() function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Example 4: Read from a binary file using fread()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
      printf("Error! opening file");
      // Program exits if the file pointer returns NULL.
      exit(1);
  }
  for(n = 1; n < 5; ++n)
  {
     fread(&num, sizeof(struct threeNum), 1, fptr);
     printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
  }
  fclose(fptr);

  return 0;
}
```

In this program, you read the same file program.bin and loop through the records one by one. In simple terms, you read one threeNum record of threeNum size from the file pointed by *fptr into the structure num.

Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record. This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek(). As the name suggests, fseek() seeks the cursor to the given record in the file.

Syntax of fseek()

```
fseek(FILE * stream, long int offset, int whence)
```

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Different whence in fseek()

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;
  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
     printf("Error! opening file");
     // Program exits if the file pointer returns NULL.
     exit(1);
  }

  // Moves the cursor to the end of the file
  fseek(fptr, -sizeof(struct threeNum), SEEK_END);
  for(n = 1; n < 5; ++n)
  {
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
    fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
  }
  fclose(fptr);

  return 0;
}
```

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

Error Handling

error.h that is used to locate errors using return values of the function call. In C, the function return *NULL* or *-1* value in case of any error, and there is a global variable *errno* which sets the error code/number. Hence, the return value can be used to check error while programming.

By principle, the programmer is expected to prevent the program from errors that occur in the first place, and test return values from functions.

C language provides the following functions to represent errors. These are:

- perror(): This function returns a string to pass to it along with the textual representation of current errno value.
- strerror(): This function is defined in string.h library and this method return a pointer to the string representation of the present errno value.

```
/* Divided By zero Error i.e. Exception*/#include <stdio.h>
#include <stdlib.h>

void main() {
  int ddend = 60;
  int dsor = 0;
  int q;

  if( dsor == 0){
    fprintf(stderr, "Division by zero! Exiting...\n");
    getch();
    exit(-1);
  }
  q = ddend / dsor;
  fprintf(stderr, "Value of quotient : %d\n", q);
  getch();
  exit(0);
}
```

Question Bank

Short Questions (2- 3 Marks)

1. Define structure.
2. Define union.
3. What is structure member? What is a relationship between structure and structure member?
4. Define Bit-Fields.
5. Can we read from a file and write to the same file without resetting the file pointer? If
6. not? Why?
7. What is the significant of EOF?
8. Which mode is used for opening a file for updating?
9. Which function is used to position a file at the beginning?
10. Which function gives the current position in the file?
11. Which function is used to write the data to randomly accessed file?
12. What are the uses of ftell and rewind function?
13. Why it is necessary to close a file during execution of the program?
14. Write different modes in file handling.
15. What are the basic file operations?

Long Questions (6-7 Marks)

1. Write following Differences
2. Structure Vs. Union
3. Array of structure Vs. Structure of array
4. Array Vs. Union
5. Array Vs. Structure
6. What is structure? What is difference between array of structure and structure of array?
7. Write a detailed note on structure with example.
   OR
8. What is structure? Why structure is needed? Differentiate between structure and array.
9. Explain union with example.
   OR
10. Explain union. How it is differ from structure?
11. Explain bit-fields in brief.
12. Explain Nested Structure in detail.
13. How can ‘.’ Operator is used inside a union and structure?
    OR
14. How is ‘.’ (dot) operator used in nested structure?
15. Explain Array of Structure.
16. Explain memory allocation in the case of union storage of data.

17. Write the similarities between structure and union with their syntax.
18. Create structure called “Tournament” with following fields:
    a. Tournament_no, name, no_of_terms
    b. Read the details of 5 tournaments and display the tournaments with highest and
    c. lowest number of terms
    d. Create structure “Cricket” with following fields:
    e. Player_Name, Team_Name, Average
    f. Use proper data types. Read 5 players records and display them in formatted manner.
19. Create a structure “Customer” with fields like CustNo, Name, Address, and Email. Take an array of structure and input the values for member variables and display it in
    a. proper format.
20. Create structure “Employee” with Eno, name, salary, DOJ.
    a. Read at least 5 records & display records of employees whose salary is more than
    b. 20000.
21. Describe the uses and limitations of getc and putc.
22. Explain all input output functions of file handling. Explain each with example.
    a. (fopen(), fclose(), getc(), putc(), fprintf(), fscanf(), getw(), putw(), fseek(), ftell(),
    b. rewind())
23. Explain functions to access files randomly in brief.
    a. (fseek(), ftell(), rewind(), fflush())
24. Explain Command Line Argument in brief.
25. Write following Differences
    a. getc Vs. getchar
    b. printf Vs. fprintf
    c. feof Vs. ferror
    d. Append mode Vs. Write mode
26. Define structure and explain how it differs from other data types..
27. How to initialize and access member of structure? Explain with example.
28. Write a program that shows concept of structure.
29. How to declare structure? Write uses of structure.
30. What do you mean by array of structure? Explain with example.
31. Write a program that shows the concept of array of structure.
32. What are macros? Explain with example.
33. Explain #if, #elseif, #undef and #pragma directive with example.
34. Define C processor and write features of C processors.