F.Y.B.Sc.IT-SEM 1

Programming Principles With C
(PUSIT101)

By,
Prof. Nikita Madwal

Unit V

6. Structures

7. File Management in C

6. Structures

Structure

❖ **DIFFERENCE BETWEEN C VARIABLE, C ARRAY AND C STRUCTURE:**

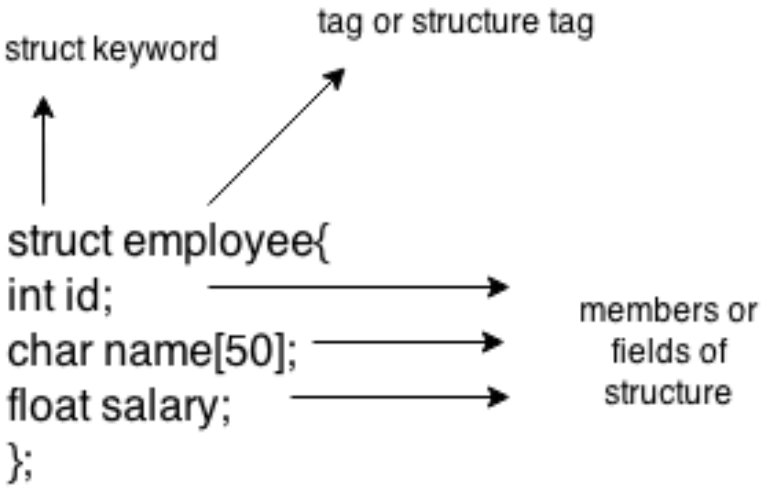
- A normal C variable can hold only one data of one data type at a time.
- An array can hold group of data of same data type.
- A structure can hold group of data of different data types and Data types can be int, char, float, double and long double etc.
- Structure is a user-defined data type in C language which allows us to combine data of different types (int ,float,char ,etc.) together under a single name.
- Structure is a collection of variables under a single name.
- Variables inside the structure are called **members** of the structure.

❖ **Declaring a structure**

➢ **struct** keyword is used to declare the structure in C.

➢ **Syntax**

```
struct structureName
{
    dataType member1;
    dataType member2;
    ...
};
```



e.g

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

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure.

❖ Declaring Structure Variables

- It is possible to **declare variables of a structure**, either along **with structure definition or after the structure** is defined.
- Structure variable declaration is similar to the declaration of any normal variable of any other data type.
- Structure variables can be declared in following two ways:

1. Declaring Structure variables separately
2. Declaring Structure variables with structure definition

- **Declaring Structure variables separately**

```
struct employee
{   int id;
    char name[50];
    float salary;
};
void main()
{
struct employee e1, e2;  //declaring variables of struct employee
}
```

- **Declaring Structure variables with structure definition**

```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

NOTE: If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

❖ Structure Initialization

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

```
struct patient
{
float height;
int weight;
int age;
};

void main()
{
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;
```

❖ Accessing members of the structure

- Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable.
- There are two ways to access structure members:
- By " ." (member or dot operator)
- By '->"(structure pointer operator or arrow operator)

e.g.

Let's see the code to access the *id* member of e1

e1 variable by "." (member operator)

e1.id

pn variable by "->" (arrow operator)

pn->id (pn is pointer variable)

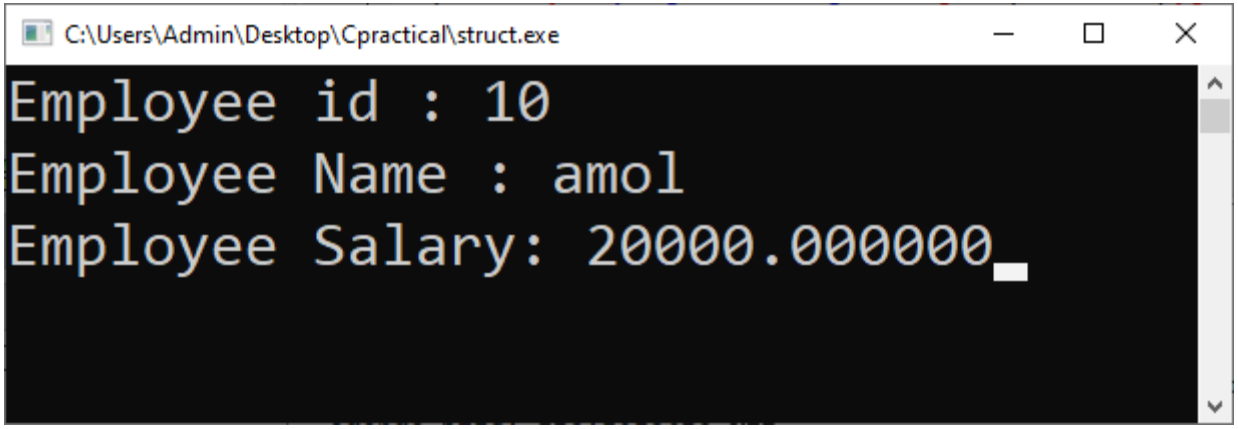
NOTE: While using the pointer in structure the pointer variable will use to access the member.

e.g.

struct.c

```c
#include<stdio.h>
struct employee //created a structure employee
{
  int id;
  char name[50]; //id,name,salary are the members of structure
  float salary;
};

void main()
{
  struct employee emp1={10,"amol",20000}; //emp1 is the variable of structure employee

  printf("Employee id : %d\n", emp1.id);  //Displaying the value of structure variable
  printf("Employee Name : %s\n",emp1.name);
  printf("Employee Salary: %f", emp1.salary);
  getch();
}
```

Output :

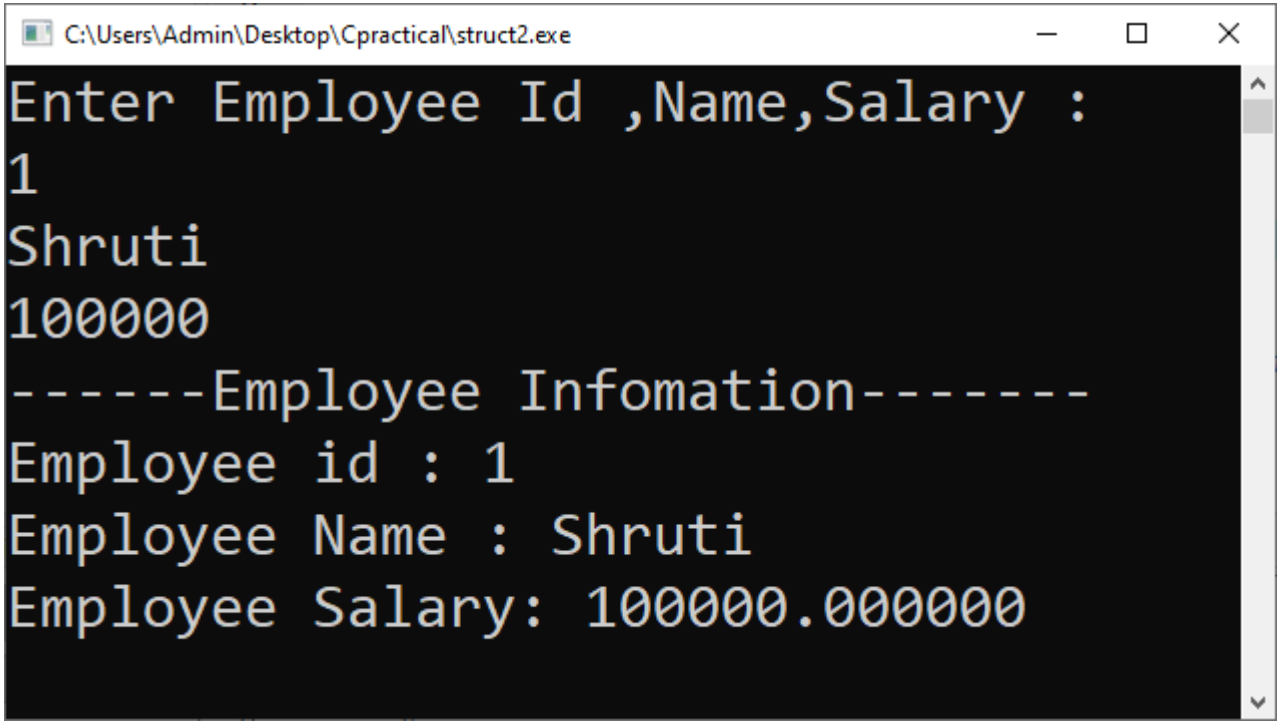

C:\Users\Admin\Desktop\Cpractical\struct.exe
Employee id : 10
Employee Name : amol
Employee Salary: 20000.000000

e.g.

struct2.c

```c
#include<stdio.h>
struct emp //created a structure emp
{
  int id;
  char name[50]; //id,name,salary are the members of structure
  float salary;
};

void main()
{
  struct emp e; //e is the variable of structure emp
  printf("Enter Employee Id ,Name,Salary : \n");
  scanf("%d",&e.id);
  scanf("%s",&e.name);
  scanf("%f",&e.salary);
  printf("------Employee Infomation-------\n");
  printf("Employee id : %d\n", e.id);  //Displaying the value of structure variable
  printf("Employee Name : %s\n",e.name);
  printf("Employee Salary: %f", e.salary);
  getch();
}
```

Output :


C:\Users\Admin\Desktop\Cpractical\struct2.exe
Enter Employee Id ,Name,Salary :
1
Shruti
100000
------Employee Infomation-------
Employee id : 1
Employee Name : Shruti
Employee Salary: 100000.000000

Nested Structure

- One structure can be nested within another structure.
- **One structure within another structure** is called as Nested structure.
- In this , the one structure has another structure as member variable.

- Nested Structure syntax

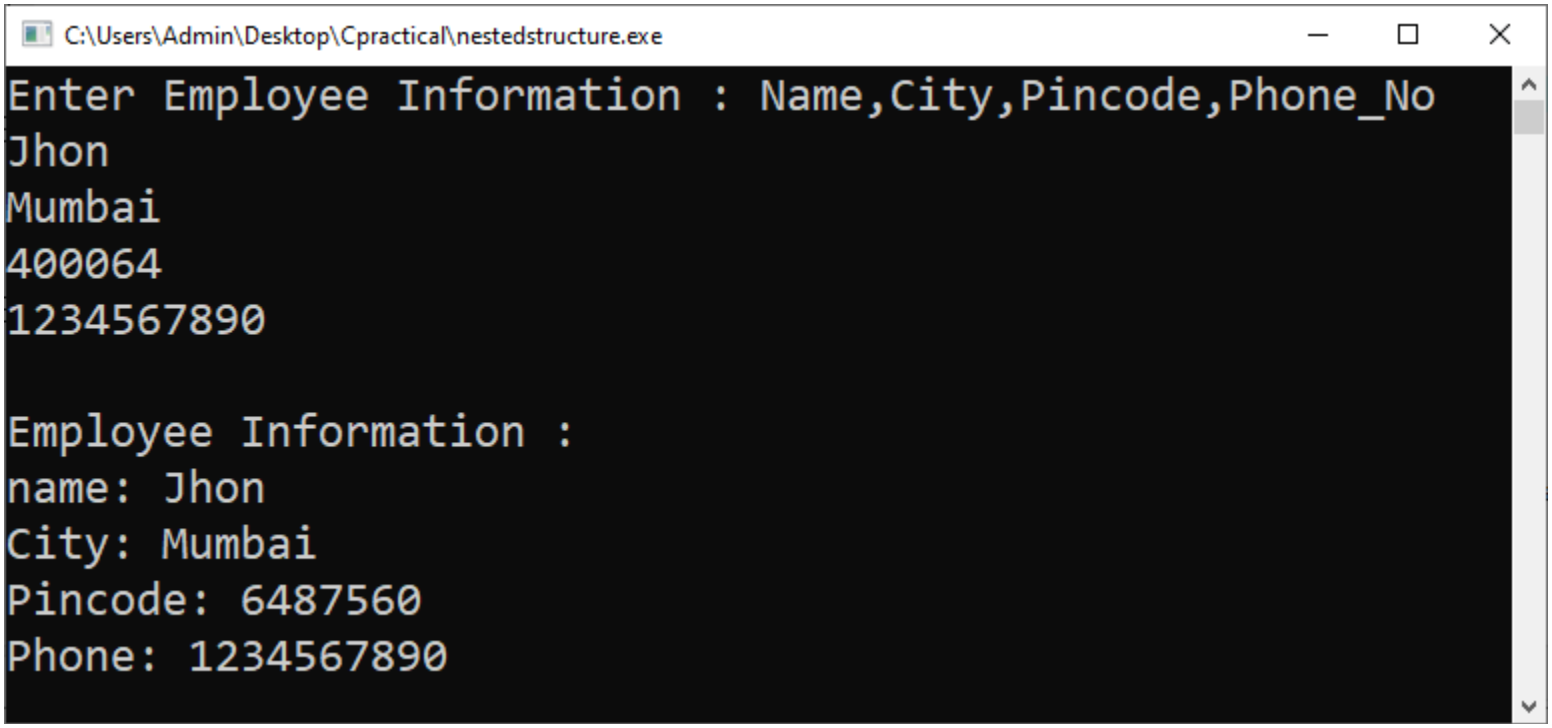
```
struct <structure_name1>
 {
    datatype member1;
    datatype member2; ...
    datatype member n;
};
struct <structure_name2>
{
    datatype member1;
    datatype member2; ...
    datatype member n;
    struct <structure_name1> stucture_variable ;
 };
```

e.g.

nestedstructure.c

```c
#include<stdio.h>
struct address
{
    char city[20];
    char pin[10];
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
void main ()
{
    struct employee emp;
    printf("Enter Employee Information : Name,City,Pincode,Phone_No\n");
    scanf("%s %s %d %s",&emp.name, &emp.add.city, &emp.add.pin, &emp.add.phone);
    printf("\nEmployee Information : \n");
    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
    getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\nestedstructure.exe
Enter Employee Information : Name,City,Pincode,Phone_No
Jhon
Mumbai
400064
1234567890
Employee Information :
name: Jhon
City: Mumbai
Pincode: 6487560
Phone: 1234567890

Structure and function

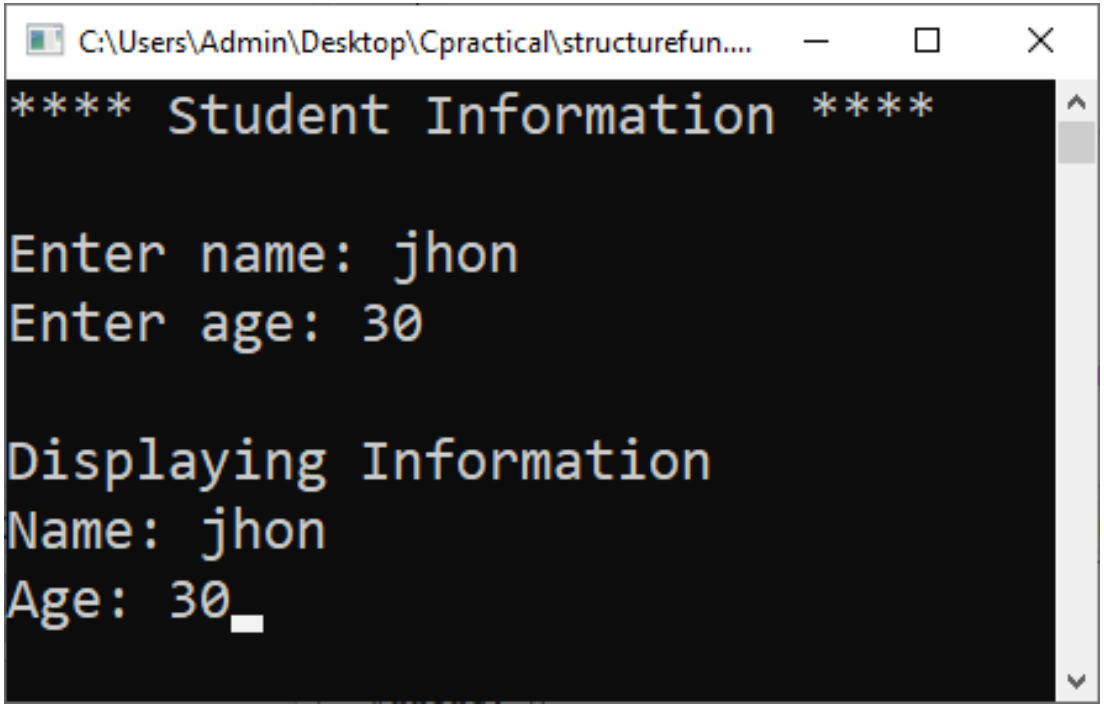
- In structure and function , we pass complete structure to function
- In this, the **structure is pass as a function argument** just like any other variable pass as a function argument.

e.g.

structurefun.c

```c
#include<stdio.h>
struct student
{
    char name[50];
    int age;
};

void display(struct student s); //function prototype

void main()
{
    struct student s;
    printf("**** Student Information ****\n\n");
    printf("Enter name: ");
    scanf("%s",&s.name);

    printf("Enter age: ");
    scanf("%d", &s.age);

    display(s); // function call
    getch();
}
void display(struct student s) //function definition
{
    printf("\nDisplaying Information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\structurefun....
**** Student Information ****
Enter name: jhon
Enter age: 30
Displaying Information
Name: jhon
Age: 30

Structures and Arrays

- **Structure containing array**

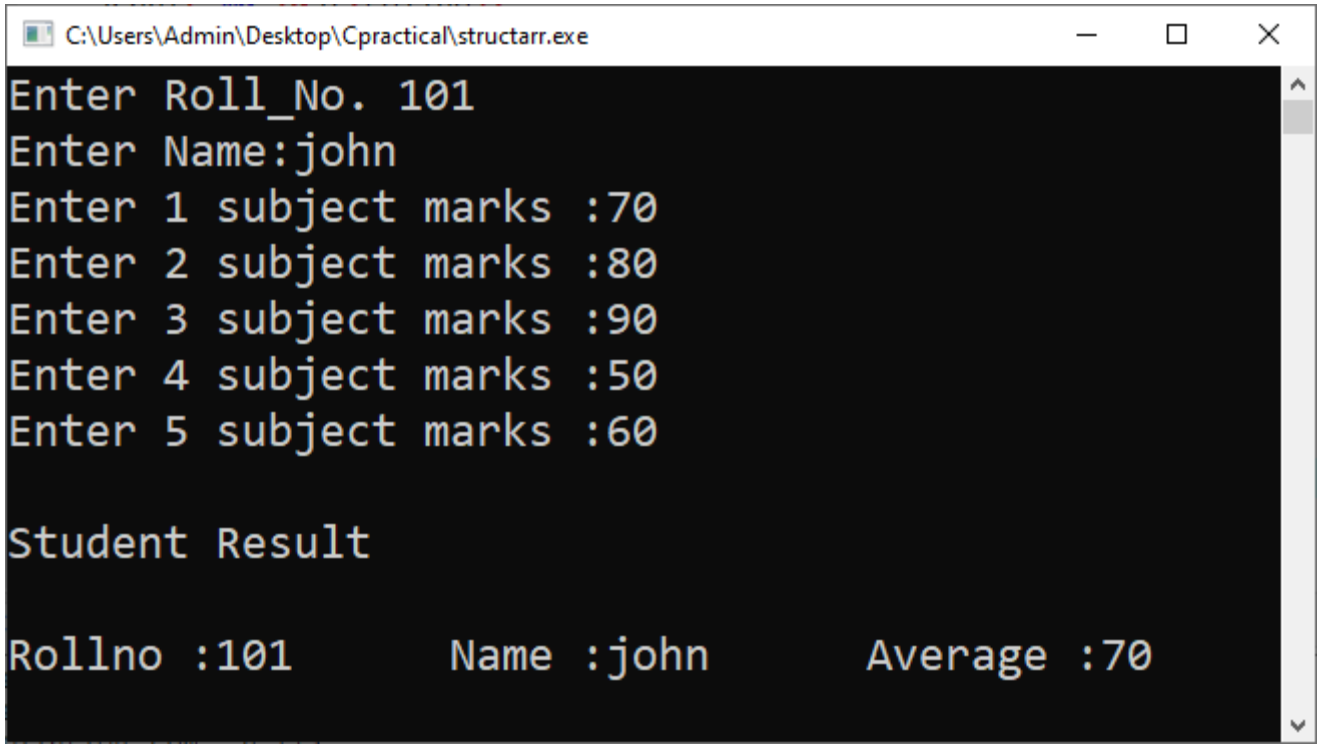
- It may also **contain an array as its member**. Such an array is called an array within a structure.
- An array within a structure is a member of the structure and can be accessed just as we access other elements of the structure.

e.g.

structarr.c

```c
#include<stdio.h>
struct student
{
    int rollno;
    int marks[5];
    char name[10];
    int avg;
};
void main()
{
    struct student st;
    int i,sum=0;
    printf("Enter Roll_No. ");
    scanf("%d",&st.rollno);
    printf("Enter Name:");
    scanf("%s",&st.name);

    for(i=0;i<5;i++)
    {
        printf("Enter %d subject marks :",i+1);
        scanf("%d",&st.marks[i]);
        sum=sum+st.marks[i];
    }
    st.avg=sum/5;
    printf("\nStudent Result\n");
    printf("\nRollno :%d\t Name :%s\t Average :%d",st.rollno,st.name,st.avg);
    getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\structarr.exe
Enter Roll_No. 101
Enter Name:john
Enter 1 subject marks :70
Enter 2 subject marks :80
Enter 3 subject marks :90
Enter 4 subject marks :50
Enter 5 subject marks :60
Student Result
Rollno :101 Name :john Average :70

➢ Arrays of Structures

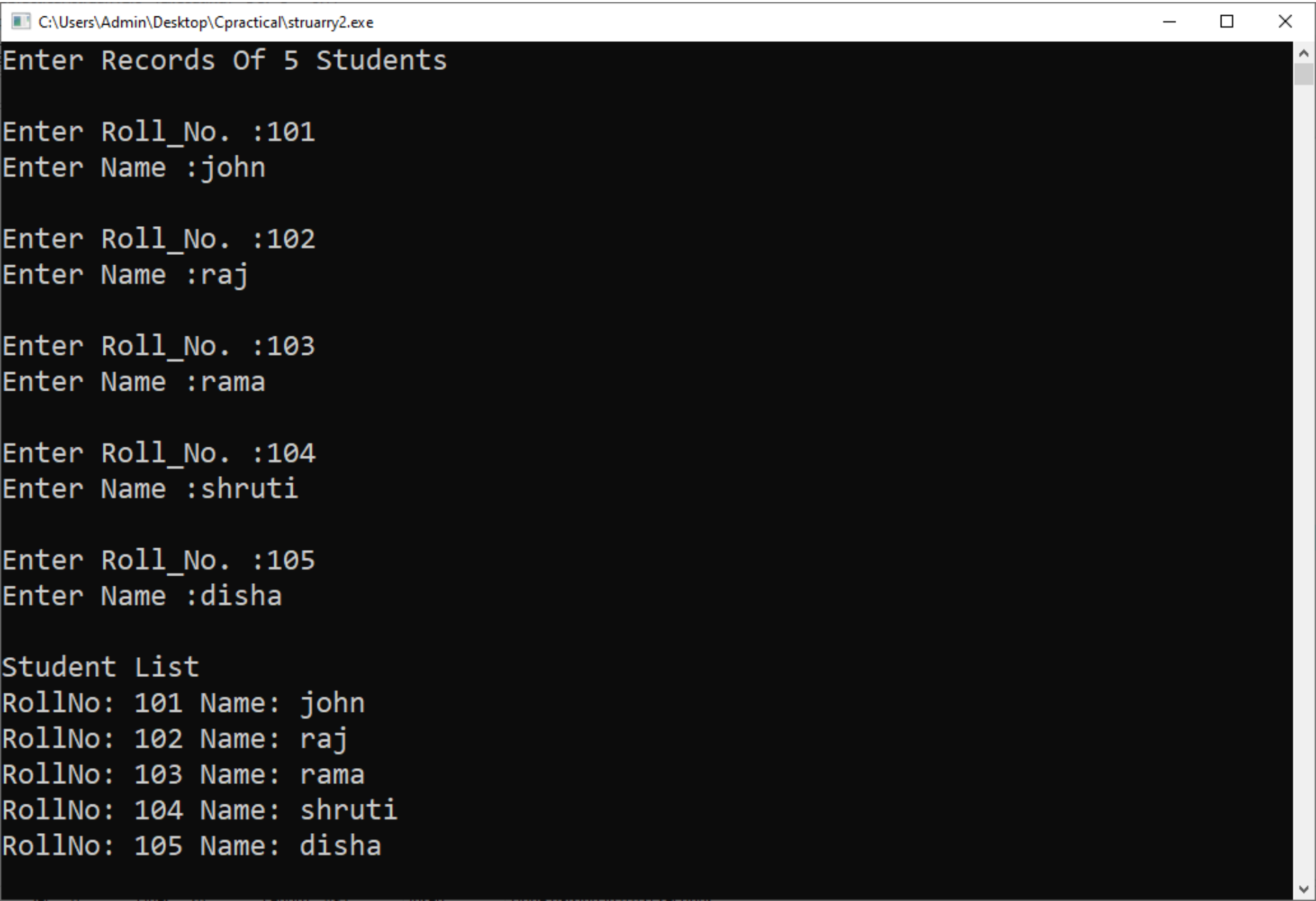
- If we need to store **more than one object** then **array of structure** is used.
- For example, to store data of 30 books we would be required to use 30 different structure variables from b1 to b30, which is definitely not very convenient.
- A better approach would be to use an array of structures.

e.g.

struarry2.c

```c
#include<stdio.h>
struct student
{
    int rollno;
    char name[20];
};
void main()
{
    struct student st[5];
    int i;
    printf("Enter Records Of 5 Students\n");
    for(i=0;i<5;i++)
    {
        printf("\nEnter Roll_No. :");
        scanf("%d",&st[i].rollno);
        printf("Enter Name :");
        scanf("%s",&st[i].name);
    }
    printf("\nStudent List");

     for(i=0;i<5;i++)
    {
        printf("\nRollNo: %d Name: %s",st[i].rollno,st[i].name);
    }
    getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\struarry2.exe
Enter Records Of 5 Students
Enter Roll_No. :101
Enter Name :john
Enter Roll_No. :102
Enter Name :raj
Enter Roll_No. :103
Enter Name :rama
Enter Roll_No. :104
Enter Name :shruti
Enter Roll_No. :105
Enter Name :disha
Student List
RollNo: 101 Name: john
RollNo: 102 Name: raj
RollNo: 103 Name: rama
RollNo: 104 Name: shruti
RollNo: 105 Name: disha

Structures and pointers

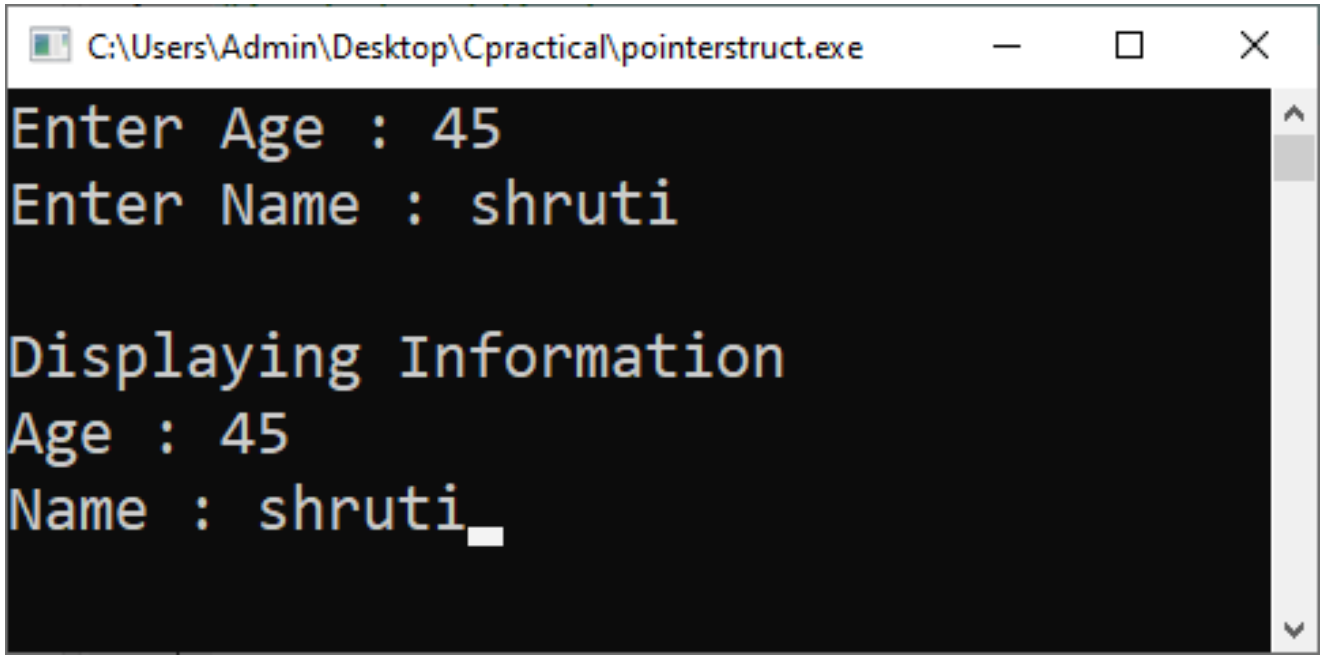
- A structure pointer is a pointer that refers to the structure. **Pointer variable contains the address of structure**
- Arrow operator(->) is used to access the elements of the structure

e.g.

pointerstruct.c

```c
#include<stdio.h>
struct person
{
    int age;
    char n[10];
};

void main()
{
    struct person p,*pn;
    pn = &p;

    printf("Enter Age : ");
    scanf("%d", &pn->age);

    printf("Enter Name : ");
    scanf("%s", &pn->n);

    printf("\nDisplaying Information\n");
    printf("Age : %d\n", pn->age);
    printf("Name : %s", pn->n);
    getch();
}
```

Output :

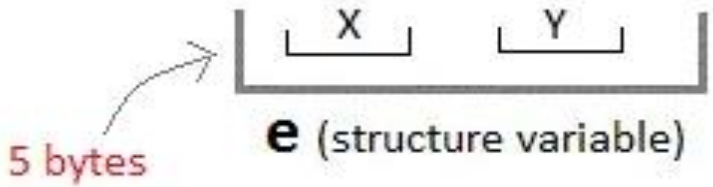

C:\Users\Admin\Desktop\Cpractical\pointerstruct.exe
Enter Age : 45
Enter Name : shruti
Displaying Information
Age : 45
Name : shruti

Unions

- **Union** can be defined as a **user-defined data type** which is a **collection of different variables of different data types** in the **same memory location**.
- **Unions is similar to structure** as structure also store different types of elements
- Syntax of structure and union are same but **major difference** between structure and union is "**memory storage**"
- However, the members within a **union** all share the same storage area within the computer's memory, whereas each member within a **structure** is assigned its own unique storage area.
- It is declared using the keyword "**union**"

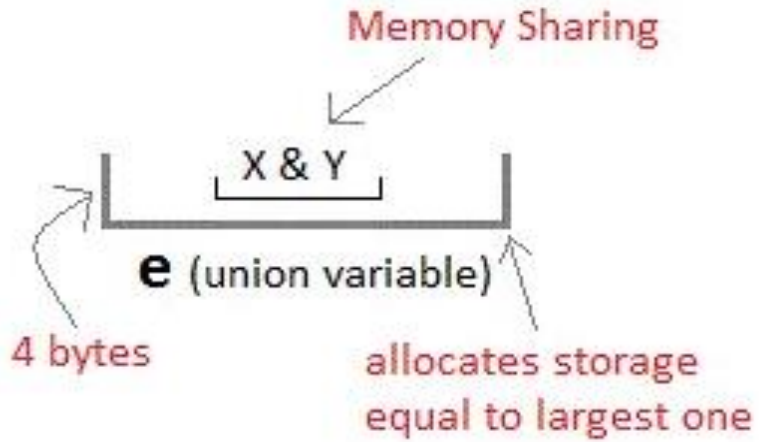
Structure

```
struct Emp
{
 char X ;     // size 1 byte
 float Y ;    // size 4 byte
} e ;
```



Unions

```
union Emp
{
 char X ;
 float Y ;
} e ;
```



e.g.

unionstruct.c

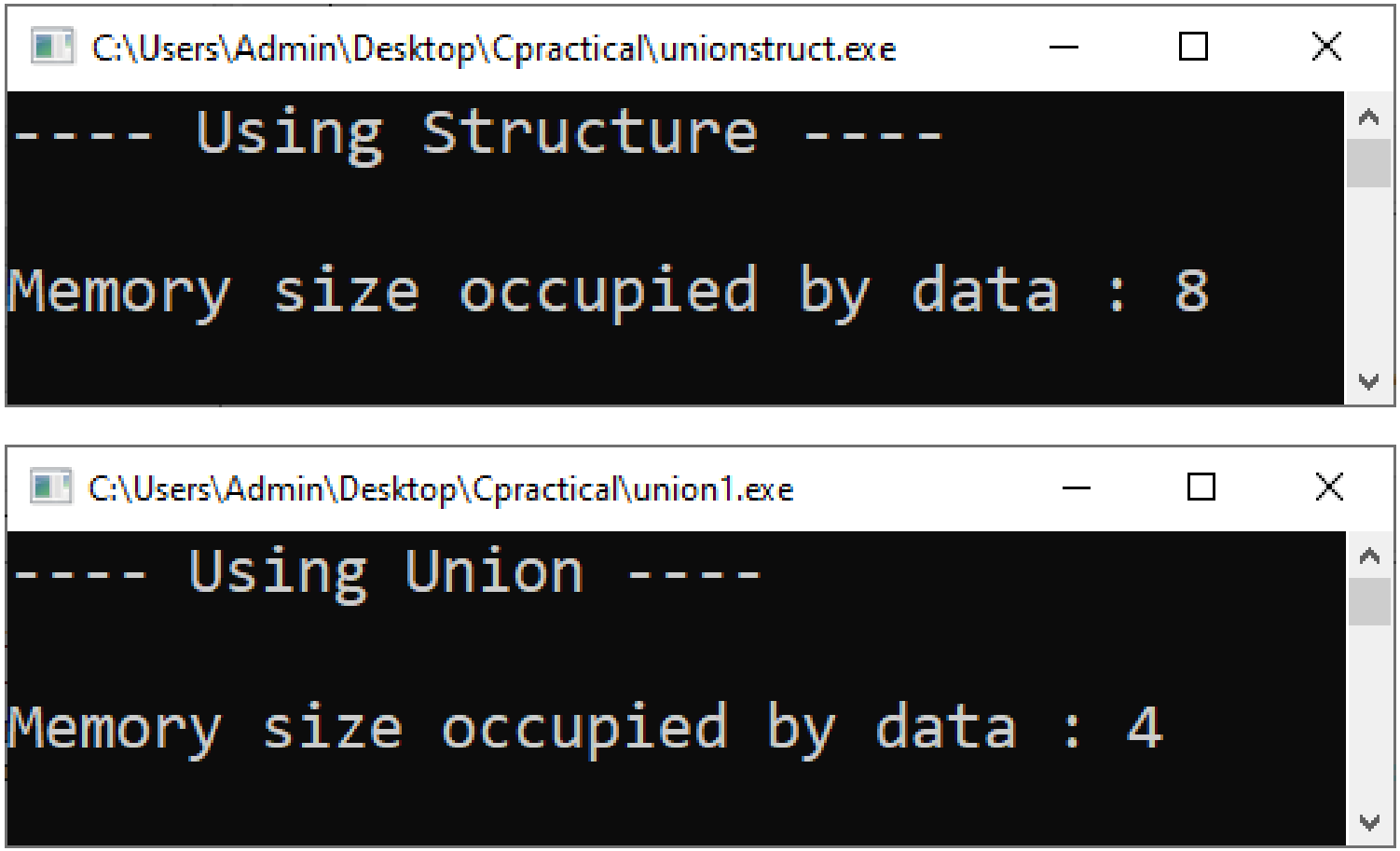
```c
#include <stdio.h>
struct Data
{
    int i;
    float str;
};

void main( )
{

    struct Data d;
    printf("---- Using Structure ----\n\n");
    printf( "Memory size occupied by data : %d\n", sizeof(d));
    getch();
}
```

union1.c

```c
#include <stdio.h>
union data
{
    int i;
    float str;
};

void main( )
{
    union data d;
    printf("---- Using Union ----\n\n");
    printf( "Memory size occupied by data : %d\n", sizeof(d));
    getch();
}
```

Difference between structure and union memory size

Output:



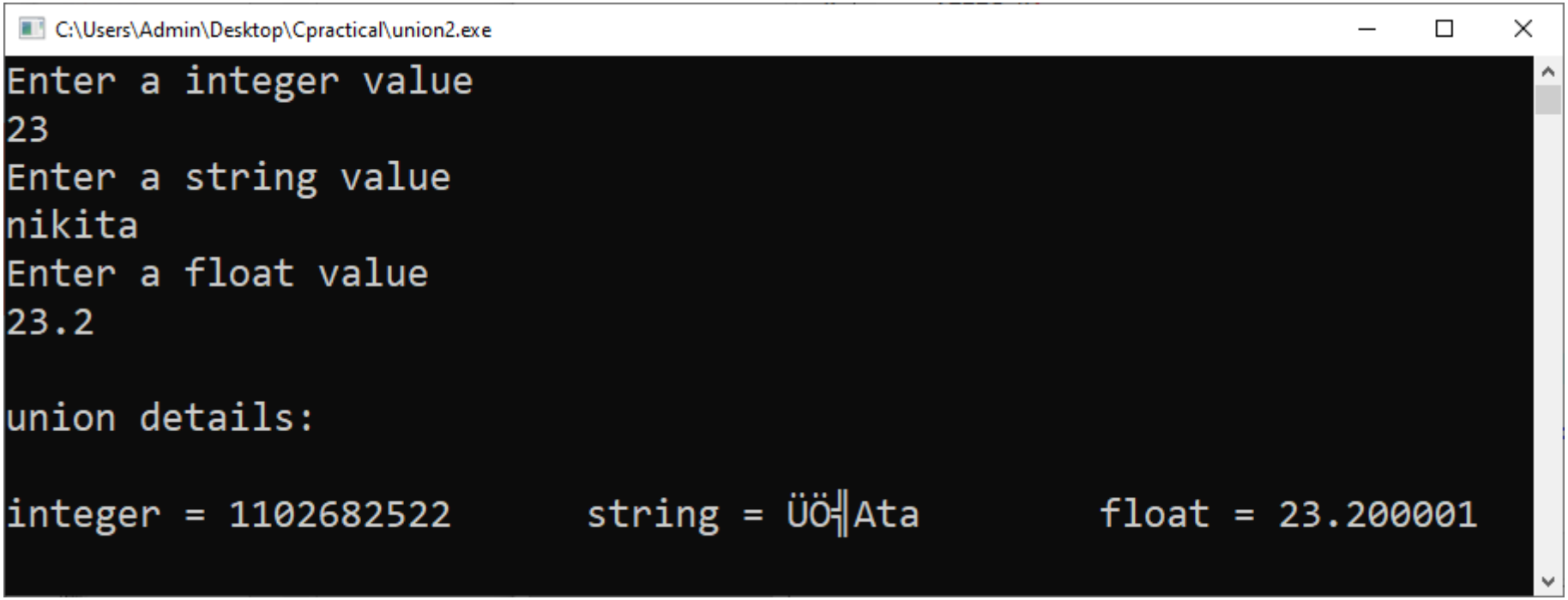
NOTE : The **sizeof** operator is the most common operator in C. It is a compile-time unary operator and used to compute the size of its operand. It returns the size of a variable.

e.g.

union2.c

```c
#include<stdio.h>
#include<conio.h>
union item
{
    int a;
    float b;
    char st[9];
};
void main( )
{
    union item it;
    printf("Enter a integer value\n");
    scanf("%d",&it.a);
    printf("Enter a string value\n");
    scanf("%s",&it.st);
    printf("Enter a float value\n");
    scanf("%f",&it.b);

    printf("\nunion details: \n");
    printf("\ninteger = %d \t  string = %s\t float = %f ",it.a,it.st,it.b);
    getch();
}
```

Output:



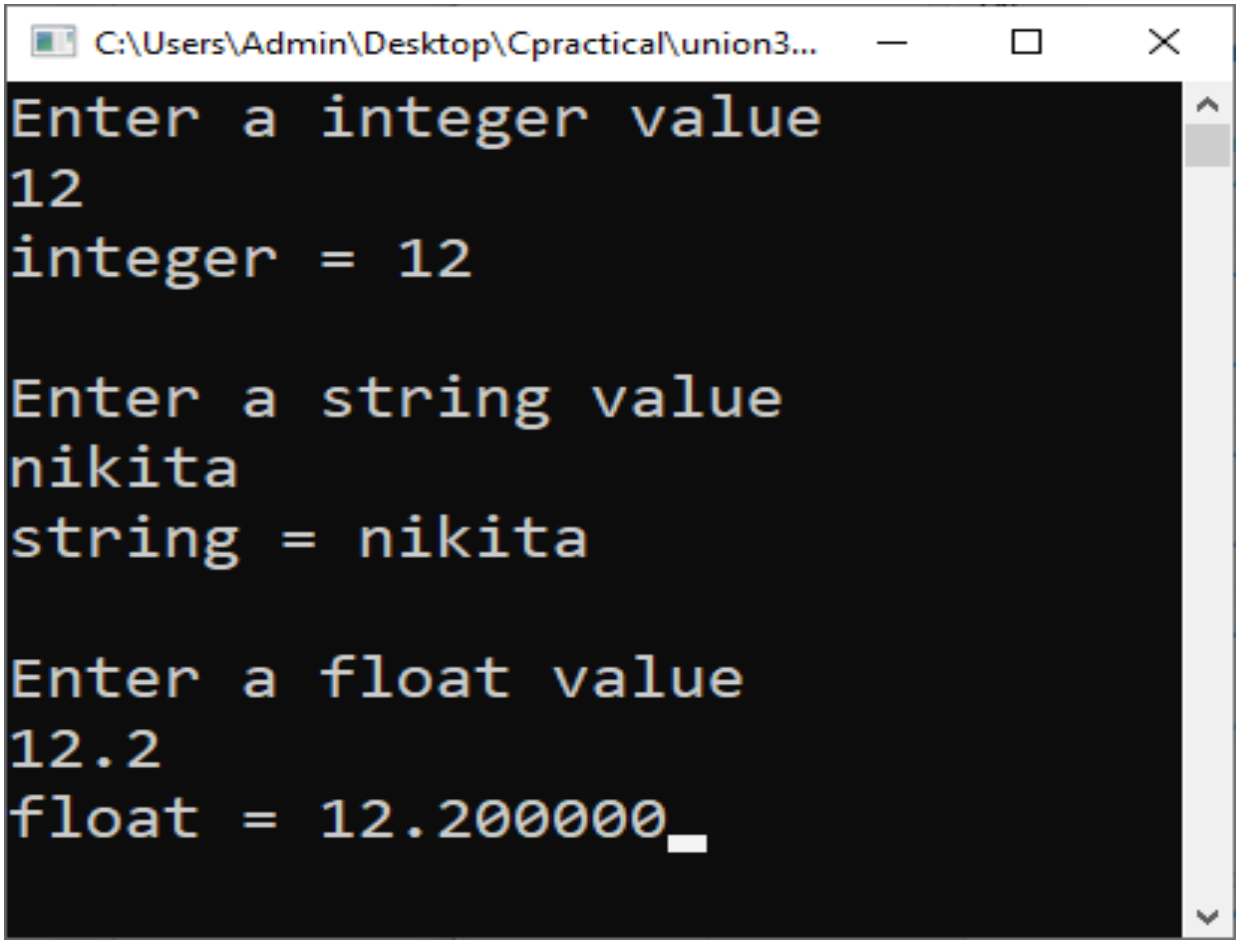
NOTE : Data gets corrupted when every value of the data type accesses the same memory location.

e.g.

union3.c

```c
#include<stdio.h>
union item
{
    int a;
    float b;
    char st[9];
};

void main( )
{
    union item it;
    printf("Enter a integer value\n");
    scanf("%d",&it.a);
    printf("integer = %d",it.a);

    printf("\n\nEnter a string value\n");
    scanf("%s",&it.st);
    printf("string = %s",it.st);

    printf("\n\nEnter a float value\n");
    scanf("%f",&it.b);
    printf("float = %f",it.b);
    getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\union3...
Enter a integer value
12
integer = 12
Enter a string value
nikita
string = nikita
Enter a float value
12.2
float = 12.200000

Bit-fields

- So far, we have been using integer fields of size 16 bits to store data, There are occasions where data items require much less than 16 bits space.
- In such cases, we waste memory space.
- Fortunately, C permits us to use small bit fields to hold data items and thereby to pack several data items in a word of memory.
- When we know that the value of a field or group of fields will never go over a threshold or is contained inside a **narrow range**, the **goal** is to use memory efficiently.

❖ Need for Bit-fields

- It enables the programmer to **assign memory** to **structures and unions** in bits in order to effectively use **computer memory**.
- used to **cut down on memory usage**.
- Simple to implement
- allows the code to be flexible.

❖Syntax:

```
struct tag-name
{
        data-type name1: bit-length;
        data-type name2: bit-length;
        . . . . . .
        . . . . . .
        data-type nameN: bit-length;
}
```

- The data-type is either **int or unsigned int or signed int**
- The bit-length is the number of bits used for the specified name.
- Note that the field name is followed by a **colon**.
- The bit-length is decided by the range of value to be stored. The largest value that can be stored is $\mathbf{2^{n-1}}$, where n is bit-length.

❖ **There are several specific points to observe:**

- There can be unnamed fields declared with size.
  Example- Unsigned : bit-length
  Such fields provide padding within the word.
- We cannot take the address of a bit field variable. This means we cannot use scanf to read values into bit fields.
- We can neither use pointer to access the bit fields.
- Bit fields cannot be arrayed.
- Bit fields should be assigned values that are within the range of their size. If we try to assign larger values, behavior would be unpredicted.

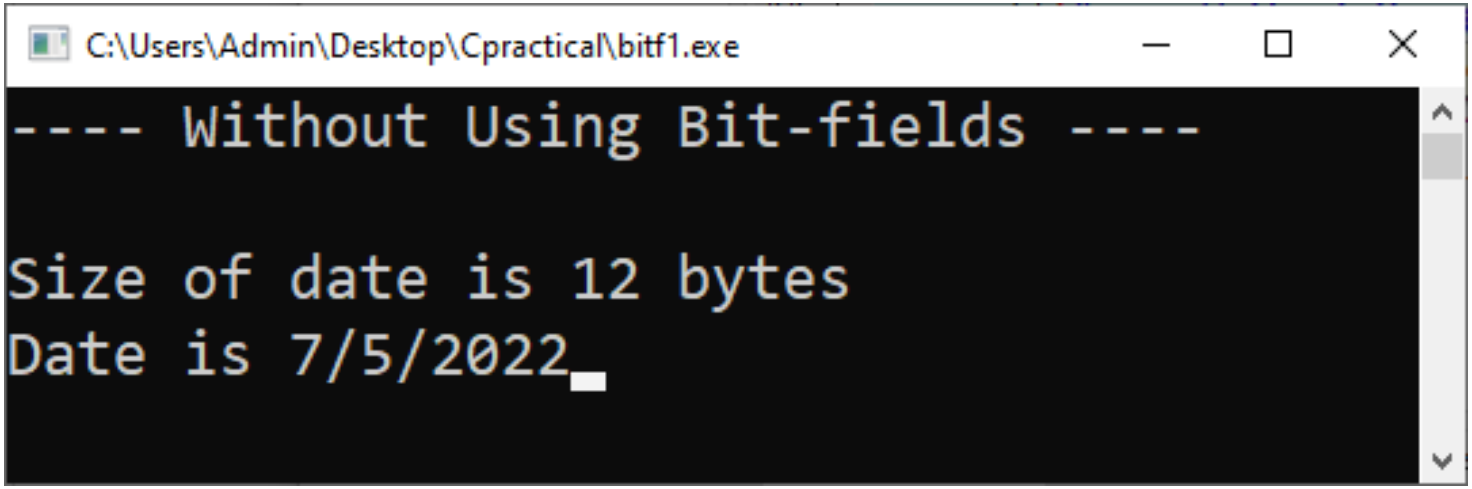
e.g.

Without using Bit-fields

bitf1.c

```c
#include <stdio.h>
// A simple representation of the date
struct date
{
  int d;
   int m;
   int y;
};
void main()
{
  printf("---- Without Using Bit-fields ----\n\n");
  printf("Size of date is %lu bytes\n",sizeof(struct date));
  struct date dt = { 7, 5, 2022 };
  printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
  getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\bitf1.exe
---- Without Using Bit-fields ----
Size of date is 12 bytes
Date is 7/5/2022

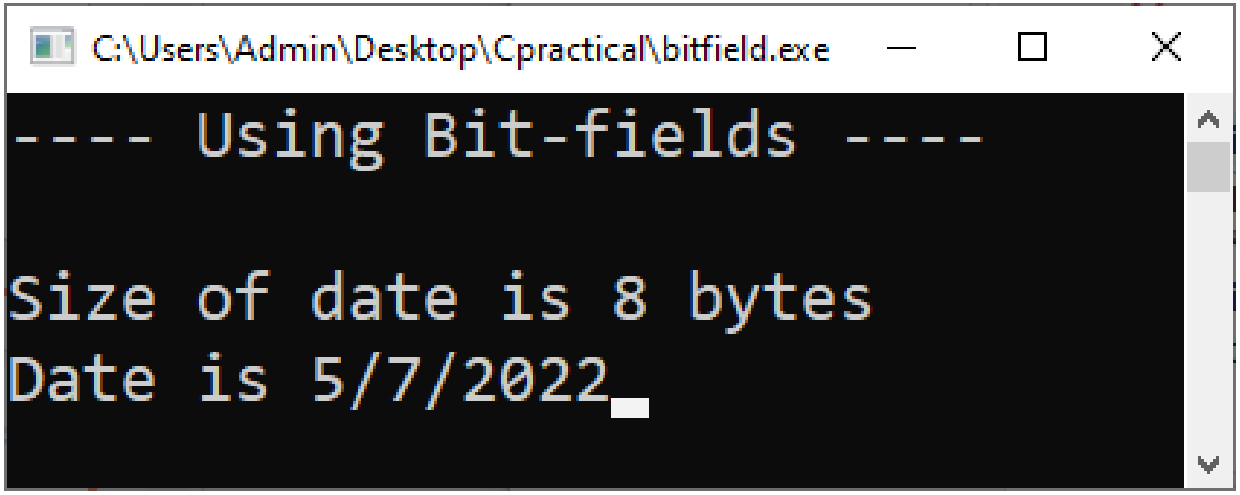
e.g.

Using Bit-fields

bitfield.c

```c
#include <stdio.h>
// Space optimized representation of the date
struct date
{

  int day : 5;      // d has value between 0 and 31, so 5 bits are sufficient
  int month : 4;  // m has value between 0 and 12, so 4 bits are sufficient
  int year;
};
void main()
{
  printf("---- Using Bit-fields ----\n\n");

  printf("Size of date is %lu bytes\n",sizeof(struct date));
  struct date dt = { 5, 7, 2022 };
  printf("Date is %d/%d/%d", dt.day, dt.month, dt.year);
  getch();
}
```

Output:


C:\Users\Admin\Desktop\Cpractical\bitfield.exe
---- Using Bit-fields ----
Size of date is 8 bytes
Date is 5/7/2022

7. File Management in C

Introduction

- 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 When a program is terminated, the entire data is lost. it is impossible to recover the programmatically generated data again and again.

- It is therefore necessary to have most flexible approach where data can be stored on the disk and read whenever necessary, without destroying the data.
- However, if we need to do so, we may **store it onto the local file system** which can be accessed every time.
- A **file** is a **collection** of related data stored on a particular area on the disk
- File helps in storing the information permanently so the information entered by the user into the file can be retrieve or use for further use.

- **Operations can be performed on a file**.
  - Naming the file
  - Opening a file
  - Reading data from the file
  - Writing to the file
  - Closing the file

- **Steps for Processing a file**
  - Declare a file pointer variable.
  - Open a file using **fopen()** function.
  - Process the file using the suitable function.
  - Close the file using **fclose()** function.

To handle files in C, file **input/output functions** available in the **stdio.h library** are:

Function	Uses/Purpose
fopen	Opens a file.
fclose	Closes a file.
getc	Reads a character from a file.
putc	Writes a character to a file.
getw	Read integer.
putw	Write an integer.
fprintf	Prints formatted output to a file.
fscanf	Reads formatted input from a file.
fgets	Read a string of characters from a file.
fputs	Write a string of characters to a file.

C **fopen()** access mode can be one of the following values:

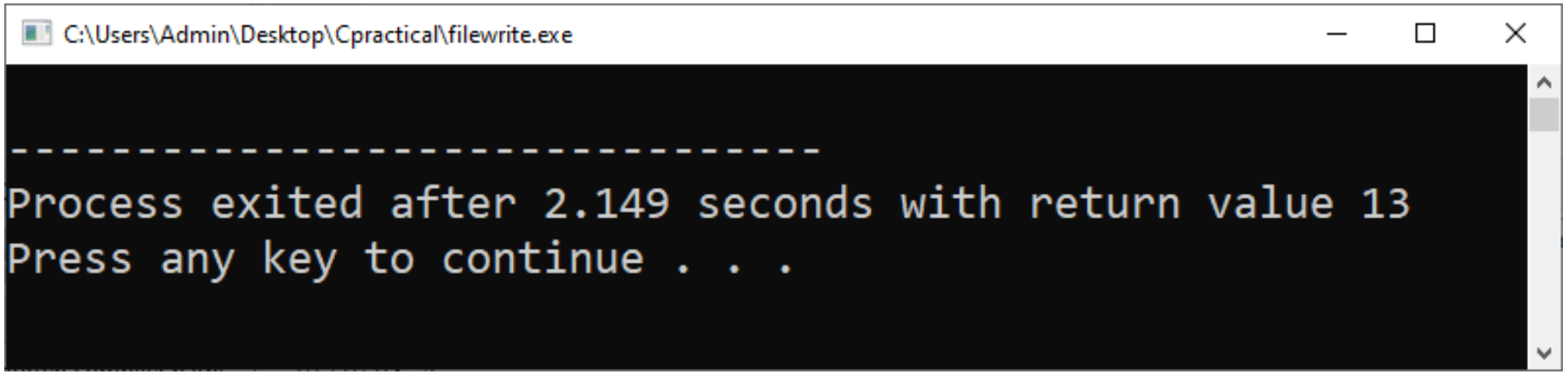
Mode	Description
r	It opens the file for reading only
w	It opens the file for writing only
a	It opens the file for appending (or adding) data to

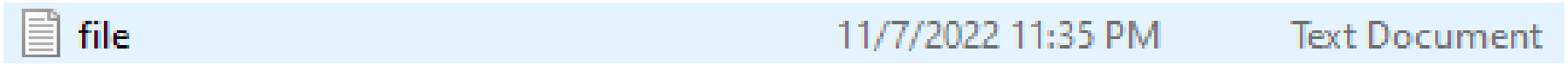
Write Mode (w)

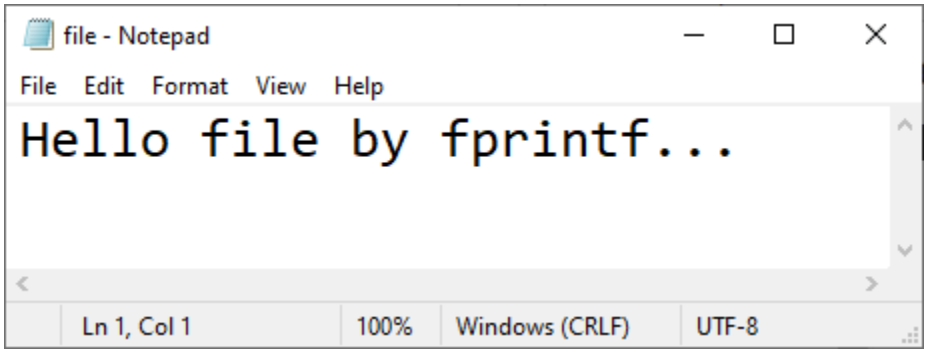
filewrite.c

```c
#include <stdio.h>
void main()
{
    FILE *fp;
    fp = fopen("file.txt", "w");  //opening file with write mode
    fprintf(fp, "Hello file by fprintf...\n"); //writing data into file
    fclose(fp);  //closing file
    getch();
}
```

Output :







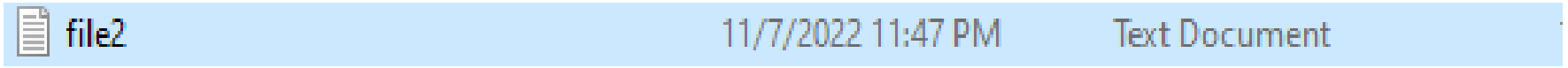
It opens the file “file.txt” , if it exists ; Otherwise it creates a new file named “file.txt” and then writes “Hello file by fprintf…” in the file

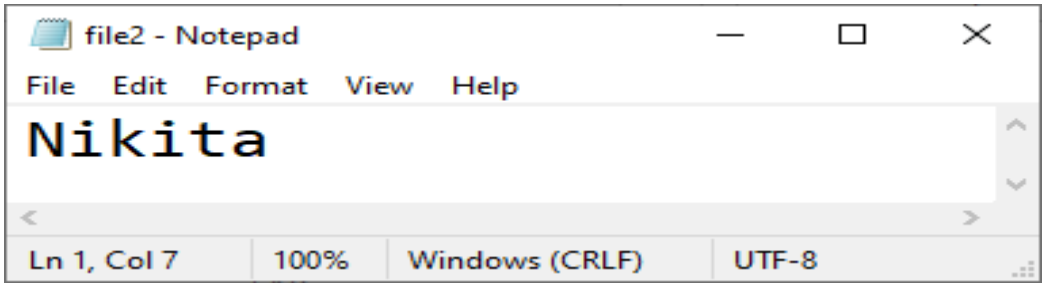
Read Mode (r)

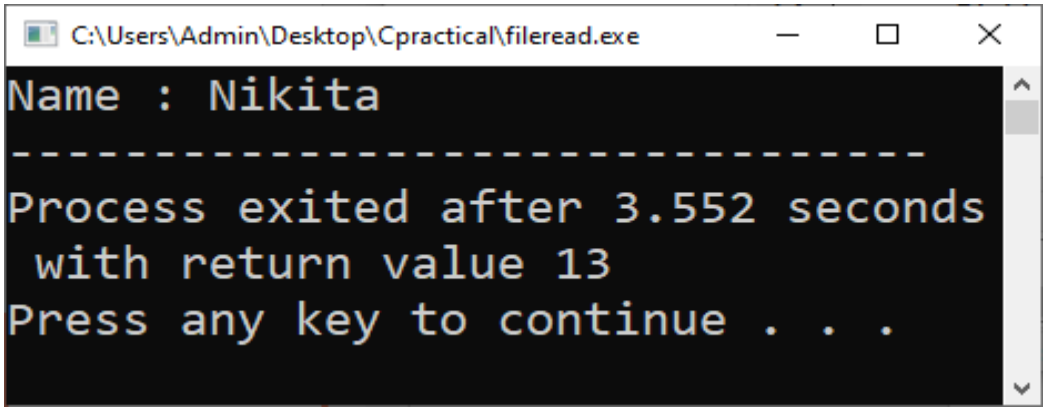
fileread.c

```c
#include <stdio.h>
void main()
{
    FILE *fp;
    char name[50];
    fp = fopen("file2.txt", "r"); //opening file with read mode

    fscanf(fp,"%s",&name);      //reading data from file
    printf("Name : %s",name);  //it will write into console
    fclose(fp);
    getch();
}
```

Output :







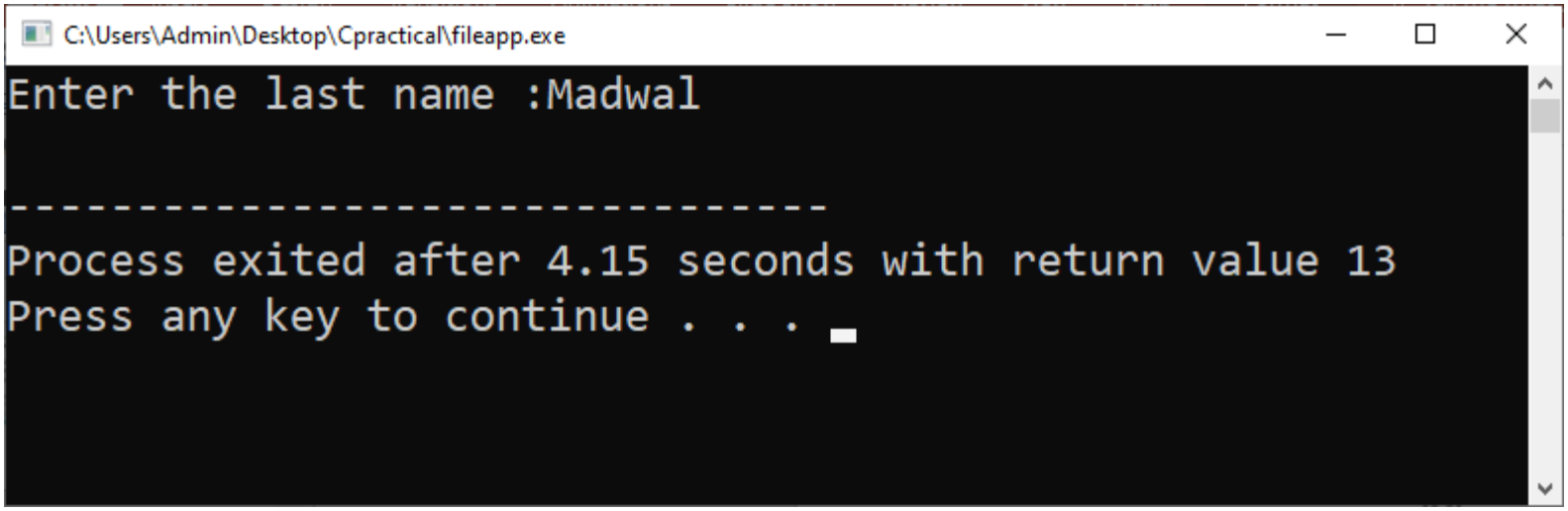
1. First text file must have some data
2. Then After writing the code in editor (Dev C++) Compile and Run.
3. The data which has been added to the text file will print to the C console.

Append Mode (a)

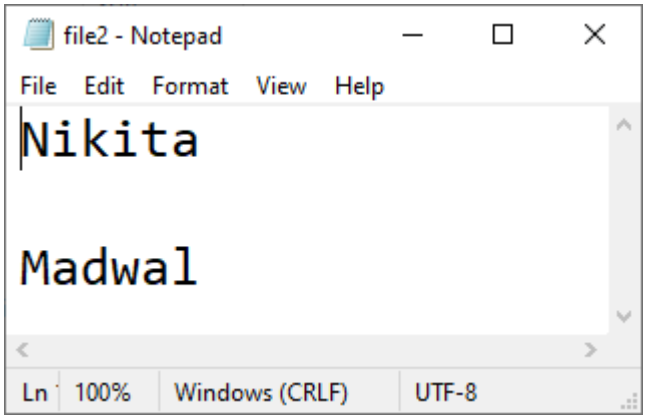
fileapp.c

```c
#include <stdio.h>
void main()
{
    FILE *fp;
    char name[50];
    fp = fopen("file2.txt", "a"); //opening file with append mode

    printf("Enter the last name :");
    scanf("%s",&name);
    fprintf(fp,"\n%s",name); //display into file
    fclose(fp);
    getch();
}
```

Output :


C:\Users\Admin\Desktop\Cpractical\fileapp.exe
Enter the last name :Madwal

Process exited after 4.15 seconds with return value 13
Press any key to continue . . .


file2 - Notepad
File Edit Format View Help
Nikita
Madwal
Ln 100% Windows (CRLF) UTF-8

Error Handling in C

- It is possible that an error occur during I/O operations on a file. Typical error situation such as (Trying to use a file that has been not opened, Device overflow, Opening a file with an invalid name)
- If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output.
- Fortunately, a few methods and variables defined in **errno.h** header file can be used to point out error using the return statement in a function.

- In C language, a function returns -1 or NULL value in case of any error and a global variable **errno** is set with the error code. So the return value can be used to check error while programming.

- **errno, perror() and strerror()**

- The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.
- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The **strerror()** function defined in **string.h** header file, which returns a pointer to the textual representation of the current errno value.

errno

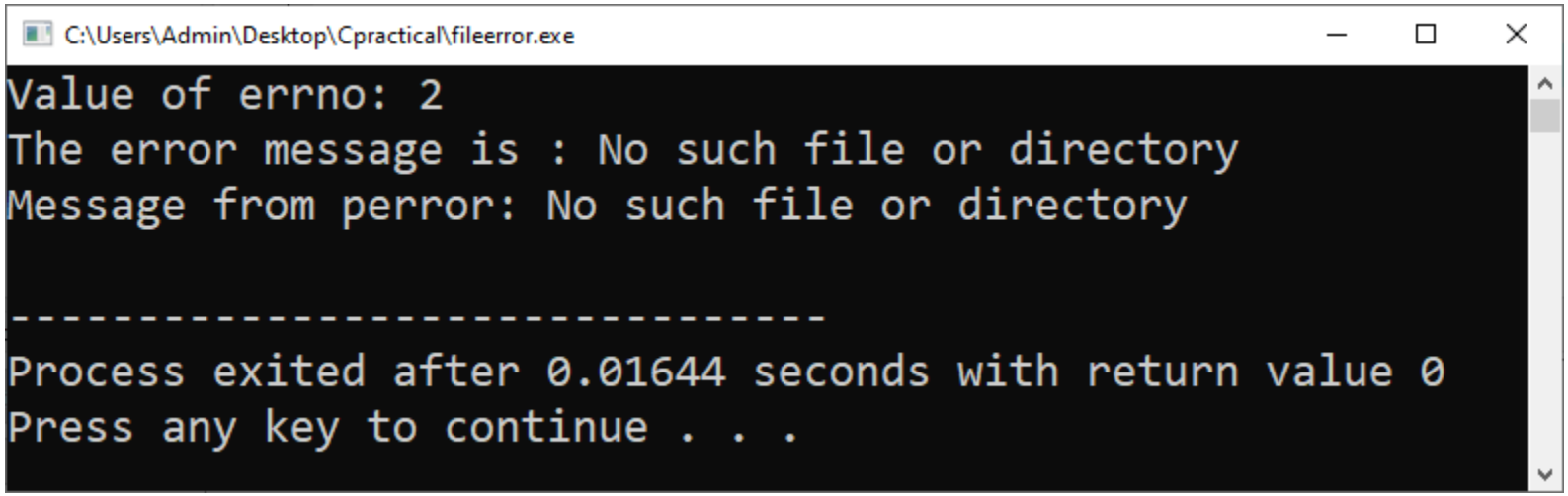
errno value	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

e.g.

fileerror.c

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
void main ()
{
    FILE *fp;

    /*If a file is opened which does not exist,
     then it will be an error and corresponding
     errno value will be set */
    fp = fopen("cprogramming.txt ", "r");

    /*opening a file which does not exist*/
    printf("Value of errno: %d\n", errno);
    printf("The error message is : %s\n",strerror(errno));
    perror("Message from perror");
    getch();
}
```

Output :


C:\Users\Admin\Desktop\Cpractical\fileerror.exe
Value of errno: 2
The error message is : No such file or directory
Message from perror: No such file or directory

Process exited after 0.01644 seconds with return value 0
Press any key to continue . . .

Random Access To Files

- There are two ways to access the data in file.

1. Sequential Access File
2. Random Access File

- Sequential Access to a data file means that the computer system reads or writes information to the file sequentially, starting from the beginning of the file and proceeding step by step.
- On the other hand, Random Access to a file means that the computer system can read or write information anywhere in the data file.

❖ Random access to files

- A **record can be accessed directly instead of accessing all records**. It takes less time to access a specific record when compared to sequential access file. This can be otherwise known as **direct access file.**
- Data can be accessed using three operations. They are **fseek( ), ftell( ) and rewind( )**

- fseek( ) – set the file pointer at the specified byte.
- ftell ( ) – return the current position value of the file pointer.
- rewind( ) – move the file pointer to the beginning of the file.

❖ Syntax:

➢ **fseek()**

fseek(file pointer, displacement, pointer position);

where,

1. file pointer which points to a file.

2. displacement determines about the direction ***either forward or backward.*** If it holds **positive** value then file pointer can be moved with **forward** direction **<u>else backward direction</u>**.

3. Pointer position contains three values:

- SEEK_SET - 0 = beginning of the file,
- SEEK_CUR - 1 = current position,
- SEEK_END - 2 = end of the file.

➢ **ftell()**

ftell(FILE *fptr); where fptr – file pointer.

➢ **rewind()**

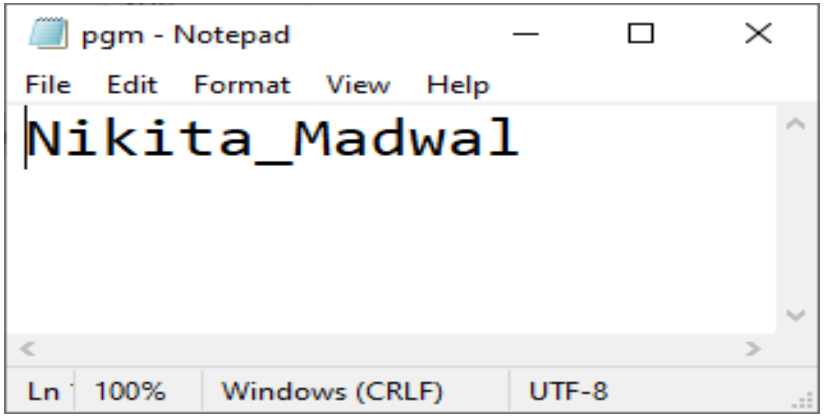
rewind(FILE *fptr); where fptr – file pointer.

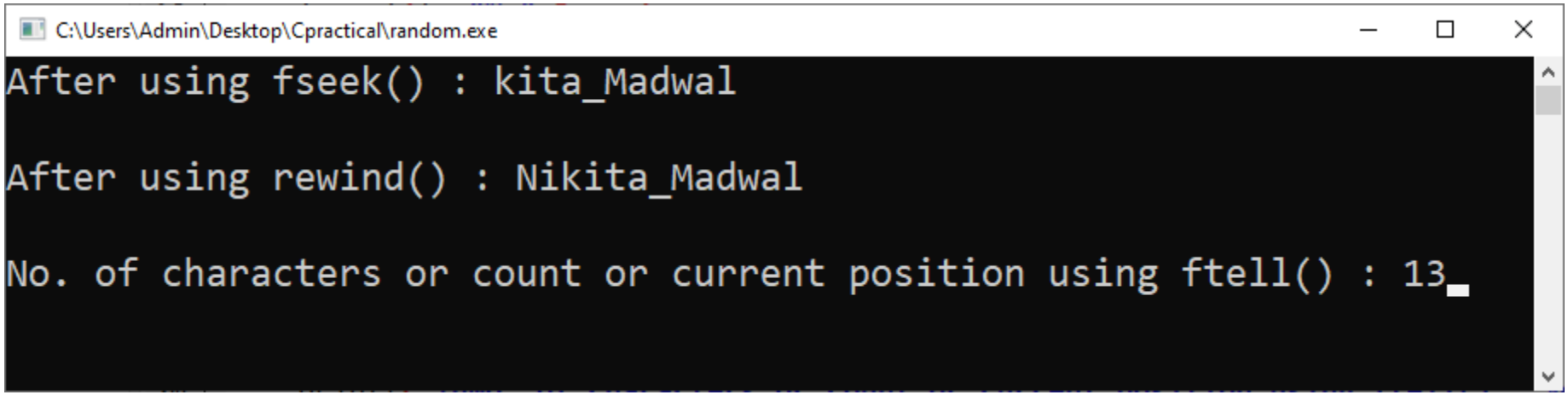
e.g.

random.c

```c
#include<stdio.h>
void main( )
{
    int len;
    FILE *fp;
    char name[10];
    fp=fopen("pgm.txt","r");

    fseek(fp,2,SEEK_SET);
    fscanf(fp,"%s",&name);
    printf("After using fseek() : %s\n",name);

    rewind(fp);
    fscanf(fp,"%s",&name);
    printf("\nAfter using rewind() : %s\n",name);

    fseek(fp,0,2);
    len=ftell(fp);
    fclose(fp);
    printf("\nNo. of characters or count or current position using ftell() : %d",len);
    getch( );
}
```

Output :


pgm - Notepad
File Edit Format View Help
Nikita_Madwal
Ln 100% Windows (CRLF) UTF-8


C:\Users\Admin\Desktop\Cpractical\random.exe
After using fseek() : kita_Madwal
After using rewind() : Nikita_Madwal
No. of characters or count or current position using ftell() : 13