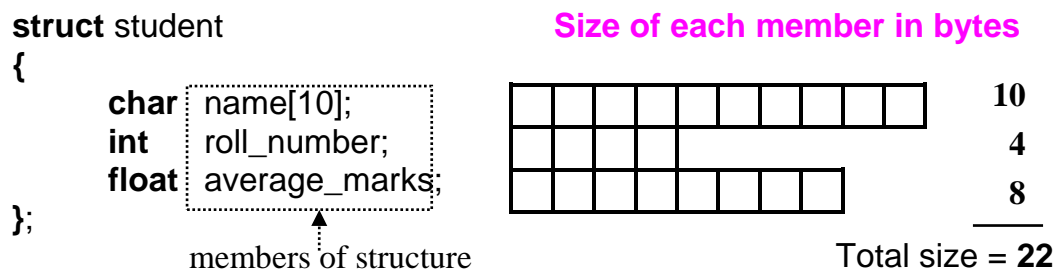


Now, we shall see “What is a structure?”

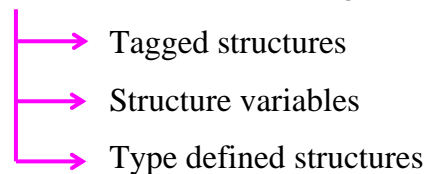
**Definition:** A *structure* is defined as a collection of data of same/different data types. All data items thus grouped are logically related and can be accessed using variables. Thus, structure can also be defined as a group of variables of same or different data types. The variables that are used to store the data are called *members of the structure* or *fields of the structure*. In C, the structure is identified by the keyword **struct**.

**Ex:** The structure definition to hold the student information such as *name*, *roll\_number* and *average\_marks* can be written as shown below:

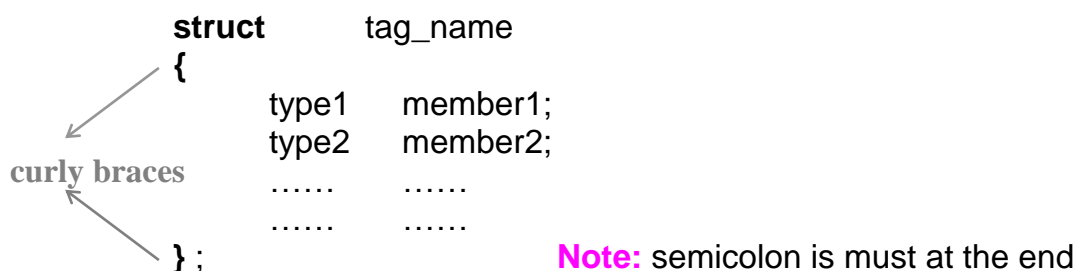


## Structure declaration

“How to declare a structure?” As variables are declared before they are used in the function, the structures are also should be declared before they are used. A structure can be declared using **three** different ways as shown below:



**Definition:** The structure definition with tag name is called **tagged structure**. The tag name is the name of the structure. The syntax of tagged structure is shown below:



For example, consider the following structure definition:

```

struct student
{
    char   name[10];           10 bytes
    int    roll_number;        4 bytes
    float   average_marks;     8 bytes
};
Total: 22 bytes

```

## Structure variables

The syntax of structure definition and declaration **using structure variables** is shown below:

```

      struct
      {
          type1  member1;
          type2  member2;
          .....
          .....
      } v1, v2, ..... vn ; Note: semicolon is must at the end of variables
                        structure variables

```

Diagram annotations: Arrows point from "curly braces" to the opening and closing curly braces of the structure definition.

For example, consider the following declaration:

```

      struct
      {
          char   name[10];           10 bytes
          int    roll_number;        4 bytes
          float   average_marks;     8 bytes
      } cse, ise;
Total: 22 bytes

```

**Type-Defined Structure:** The structure definition associated with keyword **typedef** is called **type-defined structure**. This is the most powerful way of defining the structure. This can be done using two methods:

**Method 1:** The syntax of type-defined structure is shown below:

```

      typedef struct
      {
          type1  member1;
          type2  member2;
          .....
          .....
      } TYPE_ID; Note: semicolon is must after TYPE_ID

```

Diagram annotations: Arrows point from "curly braces" to the opening and closing curly braces of the structure definition.

where

- ◆ typedef is the keyword added to the beginning of the definition
- ◆ struct is the keyword which tells the compiler that a structure is being defined.
- ◆ member1, member2,..... are called members of the structure. They are also called fields of the structure.
- ◆ The members are declared within curly braces.
- ◆ The closing brace must end with type definition name (TYPE\_ID in the syntax shown) which in turn ends with semicolon. Note that TYPE\_ID is not a variable, instead it is a user-defined data type.

For example, the type-defined structure definition is shown below:

```
typedef struct
{
    char name[10];
    int roll_number;
    float average_marks;
} STUDENT;
```

Since STUDENT is the type created by the user, it can be called as user-defined data type. From this point onwards we can use **STUDENT** as data type and declare the variables. For example, consider the following declaration:

STUDENT cse, ise ;

This statement declares that the variables *cse* and *ise* are variables of type STUDENT.

**Method 2:** Here, we use the *tag* for the structure and then we obtain the user-defined data type using the keyword **typedef**. For example, consider the structure definition:

```
/* Structure definition */
struct student          Note: student is the tag name
{
    char name[10];
    int roll_number;
    float average_marks;
};                       /* No memory is allocated for structure
*/
```

The user-defined data type can be obtained using the keyword **typedef** as shown below:

```
typedef struct student STUDENT;    /* STUDENT is user-defined data type */
```

Using the user-defined data type STUDENT, we can declare the variables as shown below:

```
/* Structure declaration */  
STUDENT cse, ise;           /* Memory is allocated for the variables */
```

## Structure initialization

The structures can be initialized various ways:

**Method 1:** Specify the initializers within the braces and separated by commas when the variables are declared as shown below:

```
struct employee  
{  
    char name[20];  
    int salary;  
    int id;  
} a = {"MONALIKA", 10950, 2001};  
      └──────────┘  
      initializers
```

### Memory representation

name	M	O	N	A	L	I	K	A	\0
salary	10950								
id	2001								

**a**

**Method 2:** Specify the initializers within the braces and separated by commas when the variables are declared as shown below:

```
/* structure definition */
```

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

**Note:** The compiler will not reserve memory for structure definition

```
/* structure declaration and initialization */  
struct employee a = {"MONALIKA", 10950, 2001};
```

## Accessing structures

The members of a structure can be accessed by specifying the variable followed by dot operator followed by the name of the member. **For example,**

consider the structure definition and initialization along with memory representation as shown below:

#### //Structure initialization

```
struct employee
{
    char name[20];
    int salary;
    int id;
} a = {"MITHIL", 10950, 2001};
```

#### Memory representation

name	M	I	T	H	I	L	\0		
salary	10950								
id	2001								

**a**

The various members can be accessed using the variable *a* as shown below:

- ◆ By specifying *a.name* we can access the name "MITHIL".
- ◆ By specifying *a.salary* we can access the value 10950
- ◆ By specifying *a.id* we can access the value of 2001

Now, the question is "How to display the various members of a structure?"  
The various values can be accessed and printed as shown below:

#### Programming statements

```
printf("%s\n", a.name);
printf("%d\n", a.salary);
printf("%d\n", a.id);
```

#### Output

```
MITHIL
10950
2001
```

Once we know how to display the members of a structure, let us see "How to read the values for various members of a structure?" We know that format specifications such as %s %d %d are used to read a string, an integer and a float. The same format specifications can be used to read the members of a structure. For example, we can read the name of an employee, the salary and id as shown below:

```
gets(a.name);
scanf("%d", &a.salary);
scanf("%d", &a.id);
```

### Internal implementation of structures

Now, let us see "What is the size of the structure?"

**Definition:** The size of a structure is defined as the sum of sizes of individual member of the structure. For example, the structure declaration along with sizes of individual data members is shown below:

### // Structure definition

```
struct employee
{
    char    name[8];
    int     id;
    char    sex;
    double  salary;
} a = {"MITHIL", 109, 'M', 9999.9};
```

### Pictorial representation

name	M I T H I L \0
id	10950
sex	M
salary	9999.9

### Bytes

8

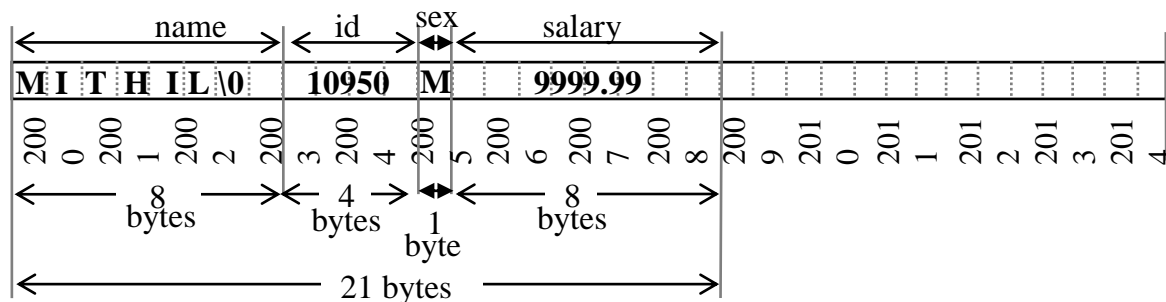
4

1

8

**Total : 21**

Observe that total size of the structure = 21 bytes. If 2000 is the starting address of the first member, then the starting address of each member depends on size of previous member. The complete memory map along with addresses is shown below:



Observe that the values of the members are stored in **increasing address locations** in the order specified in the structure definition.

That is, **address of *name* < address of *id* < address of *salary*** and the address of each member is obtained using the following relation:

**address of member = starting address of previous member + size of previous member**

So, address of member *id* = starting address of member *name* + size of member *name*  
= 2000 + 8  
= 2008

Address of member *sex* = starting address of member *id* + size of member *id*  
= 2008 + 4  
= 2012

Address of member *salary* = starting address of member *sex* + size of member *sex*  
= 2012 + 1  
= 2013

**Note:** Observe that some members have odd addresses and some members have even addresses. For example, members such as *name*, *id* and *sex* have even addresses whereas the member *salary* has odd address. In such situation, microprocessor accesses the data stored in even addresses faster than the data stored in odd addresses. So, **it is the responsibility of the compiler to allocate the memory for members such that they have even addresses so that the data can be accessed very fast. This leads to slack bytes.**

## Structure operations

**Copying of structure variables:** Copying from one structure variable to other structure variable is achieved using assignment operator provided both structures are of the same type. **For example,** consider the structure definition and declaration statements shown below:

```
struct                                struct
{                                     {
    char name[10];                    char name[10];
    int  salary;                      int  salary;
    int  id;                          int  id;
} a, b;                               } c, d;
```

**Note:** Observe that even though the members of both structures are same in number and type, both are considered to be of different structures. Hence,

```
// The following are valid           // The following are invalid
a = b;                               a = c;
b = a;                               b = d;
c = d;                               c = a;
d = c;                               d = b;
```

## Comparison of two structure variables or members

In the previous section, we have seen that copying of two structure variables of same type is allowed. But, we should remember that **comparing of two structure variables of same type or dissimilar type is not allowed**. For example, the following operations are invalid, even though *a* and *b* are of the same type.

```
a == b;    /* Invalid: Comparison is not allowed between structure variables */
a != b;    /* Invalid: Comparison is not allowed between structure variables */
```

However, the members of two structure variables of same type can be compared using relational operators. For example,

```
a.member1 == b.member2 // valid if both members have the same type
a.member1 != b.member2 // valid if both members have the same type
```

**Note:** The arithmetic, relational, logical and other various operations can be performed on individual members of structures but not on structure variables. But, if we want to compare we can do so by comparing the individual members of a structure.

## Uses of structures

Now, let us see “What is the use of structures?”

The structures can be used for the following reasons:

- ◆ Structures are used to represent more complex data structures
- ◆ Related data items of dissimilar data types can be logically grouped under a common name and all the items can be accessed using a common name.
- ◆ Can be used to pass arguments so as to minimize the number of function arguments.
- ◆ When more than one data has to be returned from the function, then structures can be used.
- ◆ Extensively used in applications involving database management
- ◆ To make the program more readable.

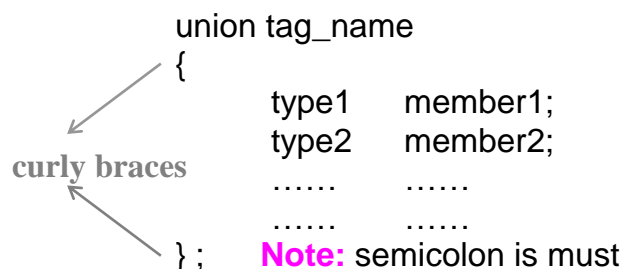
## Union and its definition

**Definition:** A *union* is similar to a structure which is also collection of data items of similar or dissimilar data types which are identified by unique names using identifiers. Each identifier is called a field or a member. All the members of the union share the same memory space. Thus, all the identifiers of **union** have the same addresses. At any given point during execution, only one member is active.

Now, let us see “How union is declared and used in C?” The general format (syntax) of a *union* definition is shown below:

```
union tag_name
{
    type1  member1;
    type2  member2;
    .....
    .....
};
```

**Note:** semicolon is must





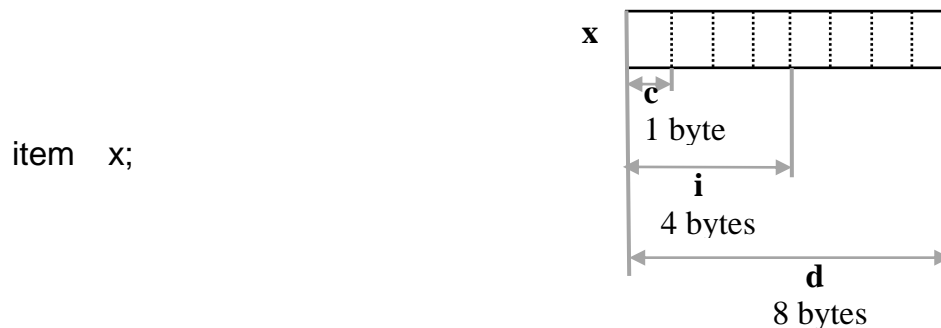
**Ex:** The union definition to hold the various informations such as *integer*, *char* and *double* values can be written as shown below:

```
// union definition
typedef union
{
    int    i;
    double d;
    char   c;
} item;
```

members of union

- ◆ Here, *i*, *d* and *c* are the **fields of the union**. They are also called **members of the union**.
- ◆ No space is reserved for the above union.

Now, consider the following declaration along with memory representation:

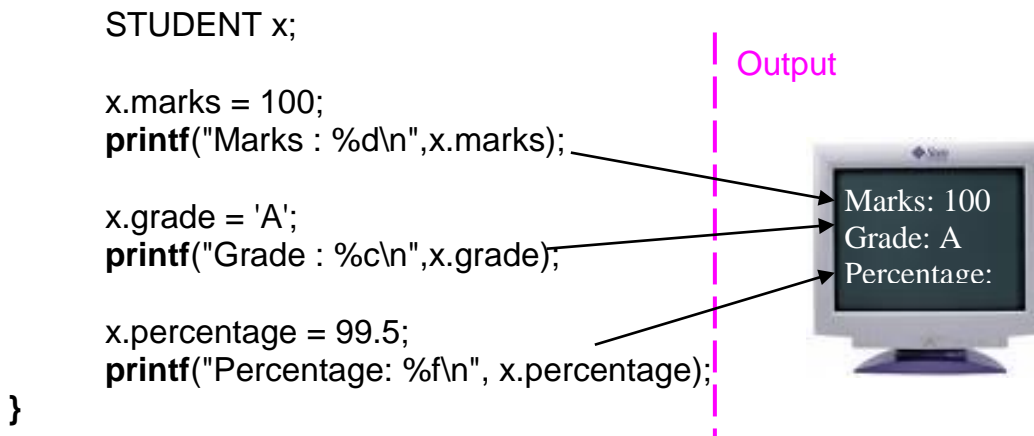


- ◆ By looking at the above declaration, the compiler will reserve the memory whose size is that of the largest member . Since **double** is the largest data type of the member of the union, `sizeof(double) = 8 bytes` of memory is reserved for the variable *x*.
- ◆ Observe that all the members have the same starting address and hence they share the same allocated space

Now, let us consider two programs to show that union and structures behave differently.

**Example :** Program to show how union behaves

```
#include <stdio.h>
void main()
{
    typedef union
    {
        int    marks;
        char   grade;
        float   percentage;
    } STUDENT;
```



Observe the following points during execution of the above program:

- ◆ After executing the statement `x.marks = 100`, the member *marks* will hold the value 100 whereas other members should not access the data. If accessed, it will be treated as garbage value.
- ◆ After executing the statement `x.grade = 'A'`, the member *grade* will hold the value 'A' whereas other members should not access the data. If accessed, it will be treated as garbage value.
- ◆ After executing the statement `x.percentage = 99.5`, the member *percentage* will hold the value 99.5 whereas other members should not access the data. If accessed, it will be treated as garbage value.

**Note:** It is observed from the above output that only one member of union can hold a value at a time. It is not possible to access all the members simultaneously. So, the variable of type `STUDENT` can be treated as *integer* variable or *char* variable or *float* variable. Now, consider the same program with structure.

---

**Example :** Program to show how structure behaves

---

```

#include <stdio.h>

void main()
{
    typedef struct
    {
        int    marks;
        char   grade;
        float  percentage;
    } STUDENT;

    STUDENT x;
}

```


```

x.marks = 100;
x.grade = 'A';
x.percentage = 99.5;

printf("Grade : %c\n",x.grade);
printf("Marks : %d\n",x.marks);
printf("Percentage: %f\n", x.percentage);
}

```

Output



**Note:** It is observed from the above output that the all the members of a structure can hold individual values at a time. It is possible to access all the members of a structure simultaneously. Now to answer the question “What is the difference between a structure and union?”

#### Structure

#### Union

1. The keyword <i>struct</i> is used to define a structure	1. The keyword <i>union</i> is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. <i>The sizeof structure is greater than or equal to the sum of sizes of its members.</i> The smaller members may end with unused <i>slack bytes</i> (see section 2.4.5)	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest member. So, <i>size of union is equal to the size of largest member</i>
3. Altering the value of a member will not affect other members of the structure	3. Altering the value of any of the member will alter other member values.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different <i>offset</i> values	4. The address is same for all the members of a union. This indicates that every member begins at <i>offset</i> zero.
5. Individual members can be accessed at any time since separate memory is reserved for each member	5. Only one member can be accessed at a time since memory is shared by each member.

# File Handling

## What are we studying in this chapter?

- ◆ Defining a file
- ◆ Opening and closing of files
- ◆ Input and output operations
- ◆ Programming examples

### 14.1 Introduction

We know that the functions **scanf()**, **gets()**, **getchar()** are used to enter the data from the keyboard and **printf()**, **puts()** and **putchar()** are used to display the result on the video display unit. This works fine when the input data is very small. But, as the volume of input data increases, in most of the applications, we find it necessary to store the data permanently on the disk and read from it. This can be done using files. Now, let us see *“What is a file?”*

**Definition:** A file is defined as a collection of data stored on the secondary device such as hard disk. An input file contains the same items we might have typed in from the keyboard. An output file contains the same information that might have been sent to the screen as the output from our program.

Now, let us see *“What are the advantages of creating and using an input file (also called data file)?”* It is very convenient to read input from a data file than to enter data interactively using the keyboard. The files are used for the following reasons:

- ◆ It is very difficult to input large volume of data through terminals
- ◆ It is time consuming to enter large volume of data using keyboard.
- ◆ When we are entering the data through the keyboard, if the program is terminated for any of the reason or computer is turned off, the entire input data is lost.

To overcome all these problems, the concept of storing the data in disks was introduced. Here, the data can be stored on the disks; the data can be accessed as and when required and any number of times without destroying the data. The data is stored on the disk in the form of a file.

#### 14.1.1 Creating a data file

Now, let us see *“How to create a data file or a text file?”*

A data file also called text file can be created as shown below:

- When we use Linux/Unix operating system, we can invoke “vi” editor
- When we are using Windows operating system, we can invoke notepad or Microsoft word.
- After invoking the editor, type the data or text and save the data or text into a file. The file along with data is stored in hard disk permanently till we physically delete it.

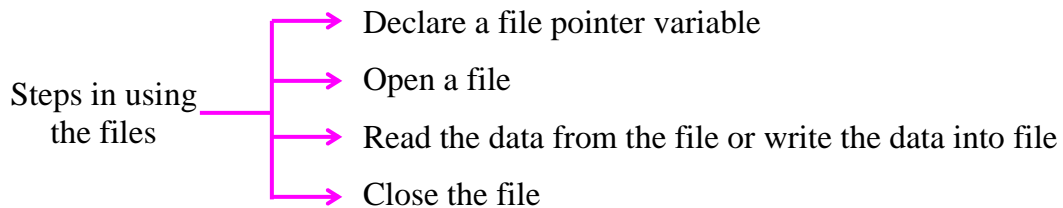
When we type the text, we may enter any sequence of characters such as letters, digits, symbols such as ; : ? # \$ and so on and insert tab characters, spaces and newline characters. All these symbols we store in a text file.

Now, let us see “What is a text file?”

**Definition:** A file that contains sequence of printable characters such as letters, digits, symbols such as ; : ? # \$ and so on including white space characters such as spaces, tabs and newline characters is called a **text file**. All the symbols that we type are stored as a stream of characters and can be processed sequentially.

- The text files are in human readable form
- They can be created and read using any text editor.
- We can read or write one character at a time.
- We can read or write one line at a time.

Now, let us see “*What are the various steps to be performed when we do file manipulations?*” The various steps to be performed when we do file manipulations are shown below:



### 14.1.2 Declare a file pointer variable

A file pointer variable should be declared using **FILE**. First, let us see “What is a FILE? Where it is defined?”

**Definition:** **FILE** is type defined structure and it is defined in a header file “stdio.h”. So, FILE can be used as a data type.

- ◆ It is derived using basic data types such as **short, char etc.**, with the help of the keyword **struct**.
- ◆ FILE is a derived data type.
- ◆ FILE is not a basic data type in C language.

Now, let us see, “How to declare a file pointer variable?” A file pointer variable can be declared using the derived data type FILE as shown below:

```
FILE *fp;
```

where

- ◆ FILE is the derived data type defined in the header file “stdio.h”. Since, FILE is defined in “stdio.h”, we have to include the file “stdio.h” in the beginning of the program.
- ◆ Since *fp* is a pointer variable, it is the responsibility of the programmer to point *fp* to the opened file.

The program segment that shows the declaration of a file pointer is shown below:

```
#include <stdio.h>

void main()
{
    FILE *fp;    /* Here, fp is a pointer to a structure FILE */

    .....      /* File operations */
    .....
}
```

**Analogy:** The information of all the students in a college is maintained by the college office. The information of a particular student such as name, address, branch and semester details along with CET ranking, percentage of marks in PUC etc. is stored in a file. If we want to read, write or update the student details we have to open a file, update a file and finally close it. The same sequence of operations can be carried out with respect to files in C also. Here too, before updating a file, it has to be opened and after updating the file, it has to be closed.

### 14.1.3 Modes of a file

Now, let us see “What are the various modes in which a file can be opened/created successfully?” The various modes in which a file can be opened successfully along with meanings are shown below:

#### Mode      Meaning

“r”	opens a file for reading. The file must exist.
“w”	creates an empty file for writing. If file does not exist, a new file is created. If a file already exists with the same name, the contents of the file are erased and the file is considered as a new empty file.
“a”	Append to a file. The data is written at the end of the existing file. If a file does not exist, a new file is created and we can start writing into the file from the beginning.
“r+”	Opens a file for both reading and writing. The file must exist.
“w+”	Creates an empty file for both reading and writing.
“a+”	Opens a file for reading and appending.

### 14.2 Opening and closing the files

Once we know various modes in which a file can be opened, let us see **how** to open a file using `fopen()`. Now, let us see “What is `fopen()`? What is its syntax?”

**Definition:** `fopen()` is a function using which

- ◆ an existing file can be opened only in read mode using mode “r”.
- ◆ an existing file can be erased and treat it as a new file. This can be done by opening the file in *write* mode using “w”.
- ◆ a new file can be created to write information into the file. This can be done by opening the file in *write mode* using “w”.
- ◆ an existing/non-existing file can be opened so that new data can be written at the end of the file. This is called *appending*. This can be done by opening the file in *append mode* using “a”.

The syntax is shown below:

```
#include <stdio.h>
FILE *fp;
.....
.....
fp = fopen(char *filename, char *mode)
```

where

- ◆ **fp** is a file pointer of type FILE
- ◆ **filename** holds the name of the file to be opened.
- ◆ **mode** can be “r” or “w” or “a”

**Return values** The function may return the following:

- ◆ A file pointer to the beginning of the opened file, if the file is opened successfully.
- ◆ NULL otherwise.

If the file pointer **fp** is not NULL, the necessary data can be accessed from the specified file. If a file cannot be opened successfully for some reason, the function returns NULL. We can use this NULL character to test whether a file has been successfully opened or not using the statement:

```
if (fp == NULL)
{
    printf("Error in opening the file\n");
    exit(0);
}
```

```
/* Using fp access file contents */
```

```
.....
.....
```

Now, let us see “What is flose()? Why it is used” flose() is a function using which

- ◆ An existing file can be closed. When we no longer need a file, we should close the file.
- ◆ This is the last operation to be performed on a file.
- ◆ This ensures that all buffers are flushed
- ◆ All the links to the file are broken.
- ◆ Once the file is closed, to access the file, it has to be re-opened.
- ◆ If a file is closed successfully, 0 is returned otherwise EOF is returned.

The syntax is shown below:

```
fclose(fp);
```

### 14.2.1 Open the file for reading

Now, let us see “How to open a file in read mode?” The steps to open a file in read mode are shown below:

- ◆ Invoke the function fopen() with two arguments
- ◆ The first argument is the file name which has to be opened



- ◆ Second argument is the mode in which the file has to be opened
- ◆ The function `fopen()` returns `NULL` if file does not exist. Otherwise, it returns pointer to the opened file.

The following set of instructions are used to open the file “input.txt”

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

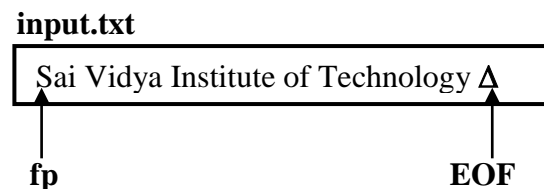
```
FILE *fp;
```

```
fp = fopen("input.txt", "r");           // Open the file in read mode
```

Now, let us see “What will happen if the non-existing file is opened in read mode? When the function `fopen()` is executed and if the file does not exist, the function `fopen()` returns `NULL`. In this situation, we display the message “Error in opening the file” as shown below:

```
if (fp == NULL)                       // If the file does not exist
{
    printf("Error in opening the file");
    exit(0);
}
```

“What will happen if the existing file is opened in read mode? If the file is existing, the function `fopen()` returns a pointer to the file which points to the beginning of the file and it is copied into file pointer variable `fp` as shown in figure below:



The final program segment to open an existing/non-existing file can be written as shown below:

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

```
FILE *fp;
```

```
fp = fopen("input.txt", "r");           // Open the file
```

```
if (fp == NULL)                       // If file does not exist
{
    printf("Error in opening the file"); // display error message
}
```

```

        exit(0);                                // terminate
    }

    /* Read the opened file and perform various operations
    .....

    fclose(fp);    // close the file

```

### 14.2.2 Open the file for writing

To open a file for writing, we use the same procedure as given in previous section. But, the file mode has to be changed from “r” to “w”. Suppose, the file “input.txt” has to be opened for writing. We can use the following instructions.

```

#include <stdio.h>

FILE *fp;
fp = fopen("input.txt", "w");                // Open the file in write mode

```

When fopen() is executed in write mode “w”, three situations may arise.

- 1) Read only file exists.
- 2) Normal file exists.
- 3) File does not exist.

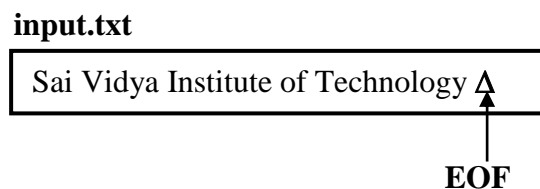
**Case 1: Read only file exists:** Now, let us see “What will happen if a read only file is opened in write mode? Suppose, the read only file “input.txt” is opened for writing. Since the file is read only, it should not be modified. So, the function fopen() returns NULL. It indicates that it is an error and the file should not be opened. The code for this can be written as shown below:

```

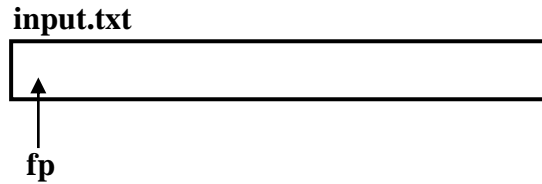
if (fp == NULL)
{
    printf("Error in opening the file");
    exit(0);
}

```

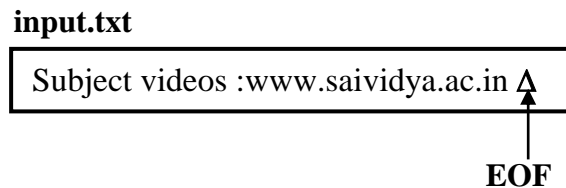
**Case 2: Normal file exists:** Now, let us see “What will happen if a normal text file which is already existing is opened in write mode? Suppose, the file “input.txt” exists as shown below:



Now, once `fopen()` is executed, the contents of the file are erased and file pointer points to the beginning of the file as shown below:

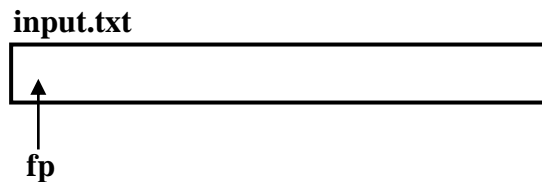


Now, we can start writing into the file from *fp* onwards as if it is a new file. Suppose, a string “Subject videos :www.saividya.ac.in” is written into the file. The contents of the file are shown below:

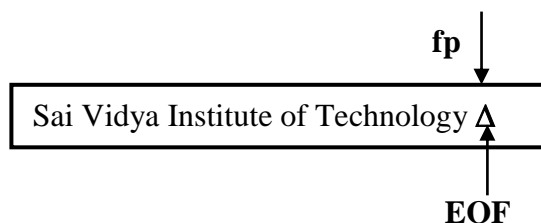


So, when a normal file is opened in write mode, the contents of the given file are deleted. The file pointer points to the beginning of the file and we can start writing the characters into the file. Now, the file pointer will always points to the end of the file.

**Case 3:** File does not exist: Now, let us see “What will happen if non-existing file is opened in write mode? Suppose, the file “input.txt” is opened for writing and the file does not exist. Now, a new file “input.txt” is created and the function `fopen()` returns a pointer to the beginning of the file and it is stored in the variable *fp* as shown below:



Now, we can start writing into the file from *fp* onwards. Suppose we write the string “Sai Vidya Institute of Technology”, then the contents of the file is shown below:



So, when a non-existing file is opened in write mode, a new file is created and we can write into the file. Now, the file pointer will always points to the end of the file. The final program segment can be written as shown below:

```
#include <stdio.h>

FILE *fp;

fp = fopen("input.txt", "w");           // Create the file

if (fp == NULL)                         // If file cannot be created
{
    printf("Error in opening the file"); // display error message
    exit(0);                             // terminate
}

// write the data into the file
.....

fclose(fp);    // close the file
```

### 14.2.3 Open the file for appending

To open a file for writing, we use the same procedure as we use in opening the file in read mode. But, the file mode has to be changed from “r” to “a”. Suppose, the file “input.txt” has to be opened for appending. We can use the following instructions.

```
#include <stdio.h>
FILE *fp;
fp = fopen("input.txt", "a");           // Open the file in append mode
```

When fopen() is executed in append mode “a”, three situations may arise.

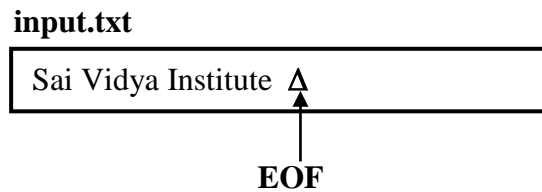
- 1) Read only file exists.
- 2) Normal file exists.
- 3) File does not exist.

**Case 1: Read only file exists:** Now, let us see “What will happen if a read only file is opened in append mode? Since the file is read only, it should not be modified. So, the function fopen() returns NULL. It indicates that it is an error and the file should not be opened. The code for this can be written as shown below:

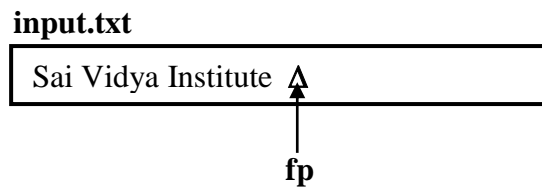
```
if (fp == NULL)
{
    printf("Error in opening the file");
    exit(0);
}
```

So, when a read only file is oopened in append mode, we cannot perform any operations on the file and hence after displaying the message “Error in opening the file” the program is terminated.

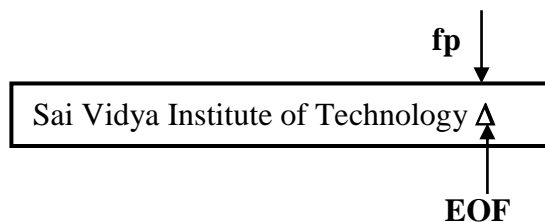
**Case 2:** Normal file exists: Now, let us see “What will happen if a normal text file which is already existing is opened in append mode? Suppose, the file “input.txt” exists as shown below:



Now, once `fopen()` is executed, the file pointer points to the end of the file as shown below:

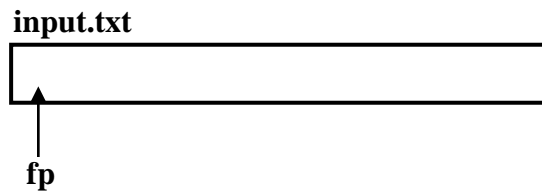


Now, we can start appending into the file from *fp* onwards as if it is a new file. The file after appending a sequence of characters “if Technology” is shown below:

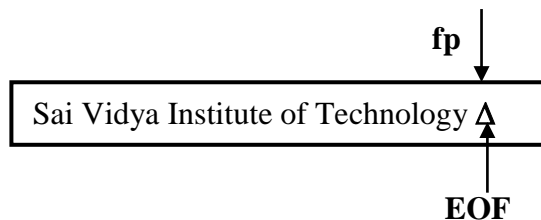


So, when a normal file is opened in append mode, we can insert the characters at the end of the file. Now, the file pointer will always points to the end of the file.

**Case 3:** File does not exist: Now, let us see “What will happen if non-existing file is opened in append mode? Suppose, the file “input.txt” has to be opened for appending and the file does not exist. Now, a new file “input.txt” is created and the function `fopen()` returns a pointer to the beginning of the file and it is stored in the variable *fp* as shown below:



Now, we can start appending into the file from *fp* onwards. When the string “Sai Vidya Institute of Technology” is written into the file, the contents of the file are shown below:



So, when a non-existing file is opened in append mode, a new file is created and we can write into the file. Now, the file pointer will always points to the end of the file as shown in above figure. The final program segment can be written as shown below:

```
#include <stdio.h>

FILE *fp;

fp = fopen("input.txt", "a");           // Open the file

if (fp == NULL)                         // If file cannot be appended
{
    printf("Error in opening the file"); // display error message
    exit(0);                             // terminate
}

// append the data into the file
.....

fclose(fp);    // close the file
```

### 14.3 I/O file functions

In this chapter we study three types of I/O functions to read from or write into the file

- ◆ File I/O functions for **fscanf()** and **fprintf()**
- ◆ File I/O functions for strings **fgets()** and **fputs()**
- ◆ File I/O functions for characters **fgetc()** and **fputc()**

### 14.3.1 fscanf(), fprintf()

Now, let us see “What is the use of fscanf() function? Explain with syntax”

**fscanf:** The function of **fscanf** and **scanf** are exactly same. Only change is that **scanf** is used to get data input from keyboard, whereas **fscanf** is used to get data from the file pointed to by **fp**. Because input is read from the file, extra parameter file pointer *fp* has to be passed as the parameter. Rest of the functionality of **fscanf** remains same as **scanf**. The syntax of **fscanf()** is shown below:

```
fscanf(fp, “format string”, list);
```

where

- ◆ **fp** is a file pointer. It can point to a **source file** or standard input **stdin**. If *fp* is pointing to **stdin**, data is read from the keyboard. If *fp* points to **source file**, the data is read from the specified source file.
- ◆ **format string** and **list** have the same meaning as in **scanf()** i.e., the variables specified in the **list** will take the values from the file specified by **fp** using the specifications provided in format string.
- ◆ The function returns **EOF** when it attempts to read at the end of the file; Otherwise, it returns the number of items read in and successfully converted.

---

**Example 14.1:** Consider the following statement:

---

```
fscanf(fp, “%d %s %f”, &id, name, &avg_marks);
```

Suppose, *fp* points to the source file. After executing above statement, the values for the variables **id**, **name** and **avg\_marks** are obtained from the file associated with file pointer **fp**. This function returns the number of items that are successfully read from the file.

---

**Example 14.2:** Consider the following statement:

---

```
fscanf(stdin, “%d %s %f”, &id, name, &avg_marks);
```

The above statement is same as the following **scanf** function:

```
scanf( “%d %s %f”, &id, name, &avg_marks);
```

Now, let us see “What is the use of fprintf() function? Explain with syntax”

**fprintf:** The function of **fprintf** and **printf** are exactly same. Only change is that **printf** is used to display the data onto the video display unit, whereas **fprintf** is used to send the data to the output file pointed to by **fp**. Since file is used, extra parameter file pointer *fp* has to be passed as parameter. Rest of the functionality of **fprintf** remains same as **printf**. The syntax of **fprintf()** is shown below:

```
fprintf(fp, "format string", list);
```

where

- ◆ **fp** is a file pointer associated with a file that has been opened for writing.
- ◆ **format string** and **list** have the same meaning as in **printf()** function i.e., the values of the variables specified in the **list** will be written into the file associated with file pointer **fp** using the specifications provided in format string.

---

**Example 14.3:** Consider the following statement:

---

```
fprintf(fp, "%d %s %f", id, name, avg_marks);
```

After executing **fprintf**, the values of the variables **id**, **name** and **avg\_marks** are written into the file associated with file pointer **fp**. This function returns the number of items that are successfully written into the file.

---

**Example 14.4:** Consider the following statement:

---

```
fprintf(stdout, "%d %s %f", id, name, avg_marks);
```

The above statement is same as the following **printf** statement:

```
printf("%d %s %f", id, name, avg_marks);
```

### 14.2.2 Read from keyboard and write into a file

Now, let us write a program to read  $n$  numbers from the keyboard and write into a file "input.dat"

---

**Example 14.5:** C program to read  $n$  numbers from keyboard and write into a file

---

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

```
void main()
```

```
{
```

```
    FILE *fp;
```

```
    int    num;
```

```
    fp = fopen("input.dat", "w");           // Open the file in write mode
```

```
    if (fp == NULL)
```

```
    {
```

```
        printf("Error in opening the file\n");
```

```
        exit(0);
```

```
    }
```



```

    while (scanf("%d", &num) != EOF)    // 10 20 30 40 50 Cntl-z (Turbo)
    {                                    //                               Cntl-d (linux)
        fprintf(fp, "%d\n", num);        // The above data is written into
    }                                    // file : input.dat

    fclose(fp);
}

```

### 14.3.2 Read from file and display on the screen

Now, let us write a program to read  $n$  numbers from the input file “input.dat” and display on the screen.

---

**Example 14.6:** C program to read  $n$  numbers from input file and display on the screen

---

```

#include <stdio.h>

void main()
{
    FILE *fp;
    int    num;

    fp = fopen("input.dat", "r");    // input.dat : 10 20 30 40 50
    if (fp == NULL)
    {
        printf("Error in opening the file\n");
        exit(0);
    }

    while (fscanf(fp, "%d", &num) > 0)
    {
        printf("%d\n", num);        // Output
                                    // 10 20 30 40 50
    }

    fclose(fp);
}

```

### 14.3.3 Read from two files and write into third file

In this section, let us write the program for the following instance of the problem. Given two university information files “*studentname.txt*” and “*usn.txt*” that contains students Name and USN respectively. Write a C program to create a new file called “*output.txt*” and copy the content of files “*studentname.txt*” and “*usn.txt*” into output file in the sequence shown below. Display the contents of output file “*output.txt*” on to the screen.

Student Name	USN
Name1	USN1
Name2	USN2
.....	.....
.....	.....

Heading

**Example 14.7:** C program to read student name and usn from two different files and write into a different file in the sequence

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

```
void main()
```

```
{
```

```
    FILE *fp1, *fp2, *fp3;
```

```
    char name[20];
```

```
    int usn;
```

```
    fp1 = fopen("studentname.txt", "r");    // Open the name file in read mode
```

```
    fp2 = fopen("usn.txt", "r");            // Open the usn file in read mode
```

```
    fp3 = fopen("output.txt", "w");        // Open the file in write mode
```

```
    for(;;)
```

```
    {
```

```
        if ( fscanf(fp1, "%s", name) > 0)    // read name from 1st file
```

```
        {
```

```
            if (fscanf(fp2, "%d", &usn) > 0)    // read name from 2nd file
```

```
            {
```

```
                fprintf(fp3, "%s %d\n", name, usn); // write to 3rd file
```

```
            }
```

```
            else break;
```

```
        }
```

```
        else break;
```

```
    }
```

```
    fclose(fp1);    /* close all the files */
```

```
    fclose(fp2);
```

```
    fclose(fp3);
```

```
}
```