# Pointer

Pointer is a derived data types. It is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

## Declaration of pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

The general form of a pointer variable declaration is

```
type *pointer-var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type

**Example**

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

## Initialization of pointer

type *pointer-var-name=&variable name;

or

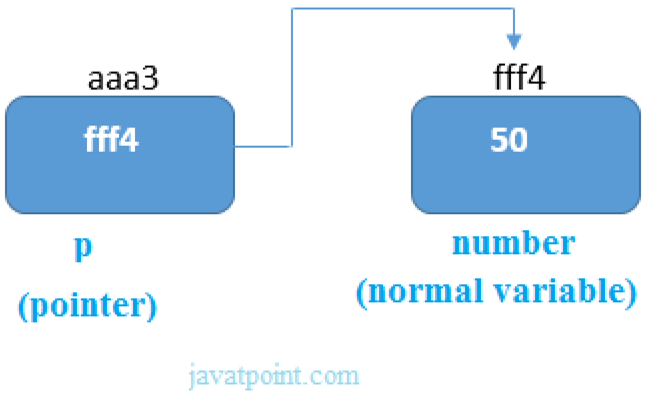
pointer-var-name=&variable name;

& operator is used to get the address of the variable.

Example

**int** number = 10;

**int*** p = &number; // Variable p of type pointer is pointing to the address of the variable n of typ e integer.



As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

* (**indirection operator**) is used to print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main()
{
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing
p gives the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer theref
ore if we print *p, we will get the value stored at the address contained by p.
getch();
return 0;
}
```

## Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

---

### Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

*32-bit*

For 32-bit int variable, it will be incremented by 2 bytes.

*64-bit*

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by
4 bytes
getch();
return 0;
}
```

*Output*

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

---

*Traversing an array by using pointer*

```
#include<stdio.h>
void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i< 5; i++)
    {
        printf("%d  ",*(p+i));
    }
   getch();
}
```

*Output*

```
printing array elements...
1  2  3  4  5
```

**Decrementing Pointer in C**

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

new_address= current_address - i * size_of(data type)

*32-bit*

For 32-bit int variable, it will be decremented by 2 bytes.

*64-bit*

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```
#include <stdio.h>
void main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immidiate prev
ious location.
getch();
}
```

*Output*

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

## Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

new_address= current_address + (number * size_of(data type))

*32-bit*

For 32-bit int variable, it will add 2 * number.

*64-bit*

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
getch();
return 0;
}
```

*Output*

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

---

## Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

new_address= current_address - (number * size_of(data type))

*32-bit*

For 32-bit int variable, it will subtract 2 * number.

*64-bit*

For 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
getch();
return 0;
}
```

*Output*

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from an another.

```
#include<stdio.h>
void main ()
{
   int i = 100;
   int *p = &i;
   int *temp;
   temp = p;
   p = p + 3;
   printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
   getch();
}
```

*Output*

```
Pointer Subtraction: 1030585080 - 1030585068 = 3
```

## **Structure**

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

**For example:** If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together that is structure.

In structure, data is stored in form of **records**.

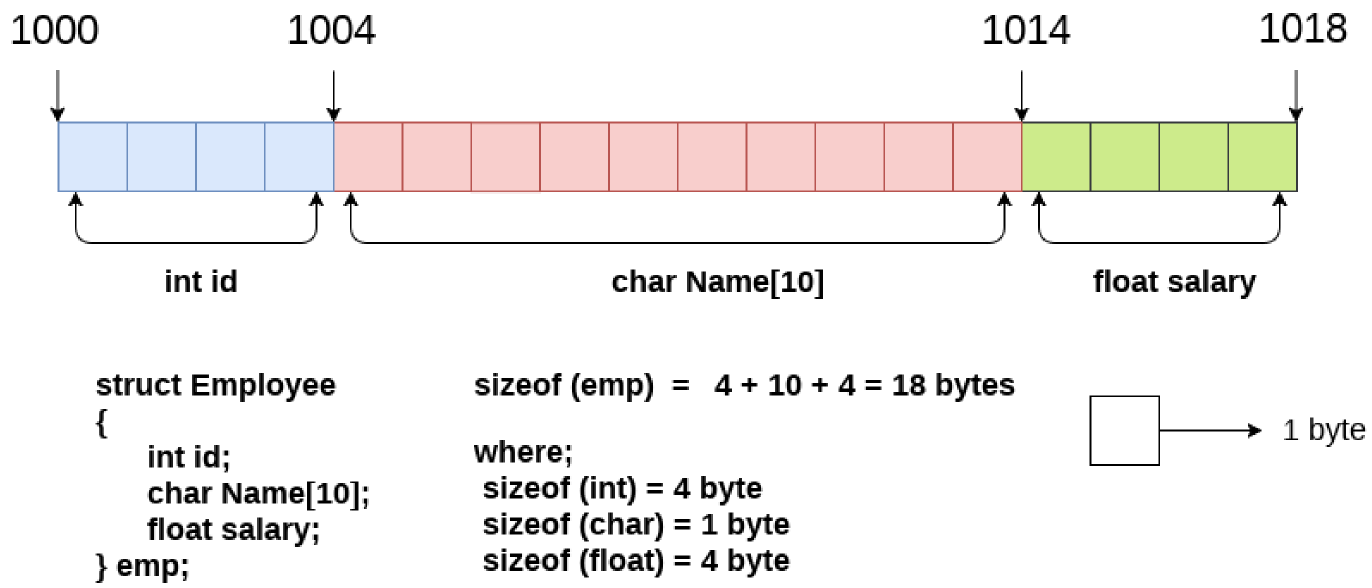
The **"struct"** keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
struct structure_name
{
data_type member1;
data_type member2;
.
.
data_type memeberN;
};
```

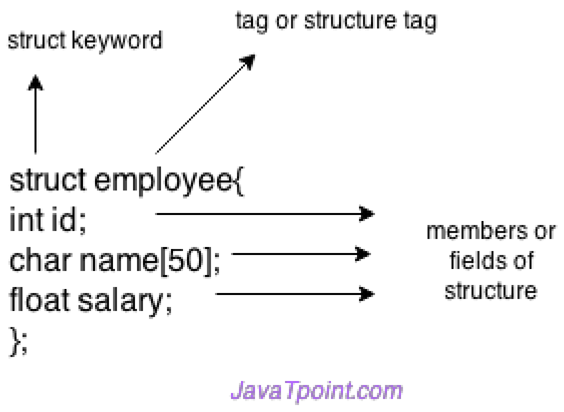
Let's see the example to define a structure for an entity employee.

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

The following image shows the memory allocation of the structure employee that is defined in the above example.



Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure. Let's understand it by the diagram given below:



## Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

**1st way:**

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

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

Now write given code inside the main() function.

```
struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

**2nd way:**

Let's see another way to declare variable at the time of defining the structure.

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

## Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. 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 using a dot . operator also called **period**or **member access** operator and -> operator also called arrow operator .

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

p1.id

## Structure Initialization

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

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

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
```

or,

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

Note: structure member can’t initialized within structure declaration

```
struct Patient
{
    float height=168.5;//error
    int weight70;  //error
    int age26; //error
};
```

## C Structure example

Let's see a simple example of structure in C language.

```
#include<stdio.h>
#include <string.h>
struct employee
{
 int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
  getch();
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
```

Let's see another example of the structure in C language to store many employees information.

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
};
```

```
int main( )
{
  struct employee e1,e2;   //declaring e1 and e2 variables for structure

   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   e1.salary=56000;

  //store second employee information
   e2.id=102;
   strcpy(e2.name, "James Bond");
   e2.salary=126000;

   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
   printf( "employee 1 salary : %f\n", e1.salary);

   //printing second employee information
   printf( "employee 2 id : %d\n", e2.id);
   printf( "employee 2 name : %s\n", e2.name);
   printf( "employee 2 salary : %f\n", e2.salary);
   getch();
   return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

## Pointers to Structures

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

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

```
struct_pointer = &structure_variable;
```

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

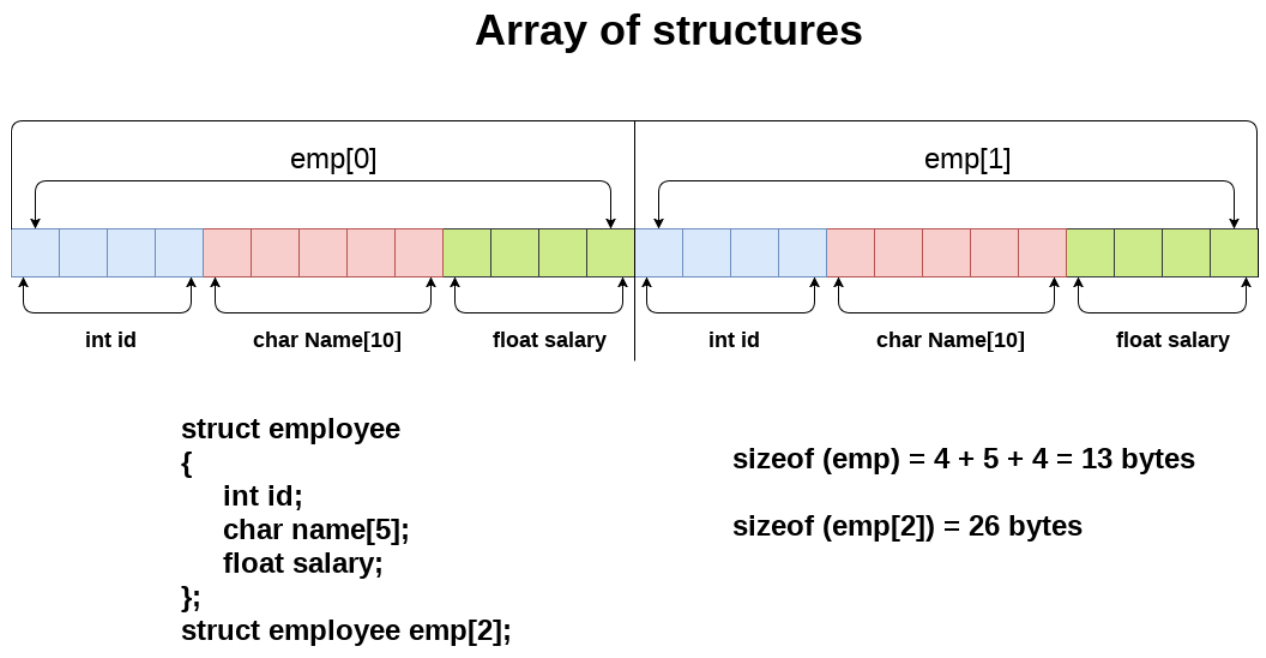
```
#include <stdio.h>
#include <string.h>

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
 {

   struct Books Book1,*book;    /* Declare Book1 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   book=&Book1;
   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
   getch();
   return 0;
}
```

Output

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
```

## Array of Structures in C

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.



```
struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];
```

Create a structure student to store the name, roll no and percentage of the student. Enter some sample values within structure and print the details of the students with maximum percentage.

```
#include<stdio.h>

struct student

{

int rollno;

char name[10];

int percentage;
```

```
};

int main()

{

int i,greater=0,n;

struct student st[5];

printf("Enter Records of 5 students");

for(i=0;i<5;i++)

{

printf("\nEnter Rollno:");

scanf("%d",&st[i].rollno);

printf("\nEnter Name:");

scanf("%s",&st[i].name);

printf("\nEnter percentage:");

scanf("%d",&st[i].percentage);

}

for(i=0;i<5;i++)

  if(st[i].percentage>greater)

   {

   greater=st[i].percentage;

   n=i;

}

   printf("details of topper student\n");

   printf("Rollno=%d\nname=%s\npercentage=%d",st[n].rollno,st[n].name,st[n].percentage);
   getch();
   return 0;
}
```

Create a structure to store and print the details of 100 books.

```
#include<stdio.h>
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
   float price;
};
int main( )
 {
   struct Books book[100];
   int i;
   for(i=0;i<100;i++)
   {
    printf("enter book name");
    scanf("%s",book[i].title);
    printf("enter author name");
    scanf("%s",book[i].author);
    printf("enter subject name");
```

```
  scanf("%s",book[i].subject);

  printf("enter book id");

  scanf("%d",&book[i].book_id);

  printf("enter book price");

  scanf("%f",&book[i].price);

 }

 printf("\n\tbook\tauthor\tsubject name\tbook\tbook price");

 for(i=0;i<100;i++)

  printf("\n\t %s \t %s \t %s \t %d \t %f", book[i].title, book[i].author, book[i].subject,
book[i].book_id, book[i].price);

 getch();

  return 0;

}
```

Let's see an example of an array of structures that stores information of 5 students and calculate percentage of each student according to given data.

```
#include<stdio.h>
struct student
{
		int rollno;
		char name[10];
		int mark[3];
		int total;
		float percentage;
};
int main()
{
		int i,j;
		struct student st[5];
		printf("Enter Records of 5 students");
```

```
            for(i=0;i<5;i++)
            {
            st[i].total=0;
            printf("\nEnter Rollno:");
            scanf("%d",&st[i].rollno);
            printf("\nEnter Name:");
            scanf("%s",&st[i].name);
            printf("enter mark of 3 subjects");
            for(j=0;j<3;j++)
            {
               scanf("%d",&st[i].mark[j]);
              st[i].total+= st[i].mark[j];
            }
            st[i].percentage= st[i].total/3;
        }
            printf("\nStudent Information List:");
            for(i=0;i<5;i++)
            {
            printf("\nRollno:%d, Name:%s, total mark:%d,perentage:%f",st[i].rollno,st[i].name,
            st[i].total, st[i].percentage);
            }
            getch();
    return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
enter mark of 3 subjects 50 45 45
Enter Rollno:2
Enter Name:Ratan
enter mark of 3 subjects 60 55 65
Enter Rollno:3
Enter Name:Vimal
enter mark of 3 subjects 45 75 60
Enter Rollno:4
Enter Name:James
enter mark of 3 subjects 65 75 70
```

```
Enter Rollno:5
Enter Name:Sarfraz
enter mark of 3 subjects 85 75 80

Student Information List:
Rollno:1, Name:Sonoo total mark:140 , percentage:46.666666
Rollno:2, Name:Ratan total mark:180, percentage:60.000000
Rollno:3, Name:Vimal total mark:180 , percentage:60.000000
Rollno:4, Name:James total mark:210, percentage:70.000000
Rollno:5, Name:Sarfraz total mark:240 , percentage:80.000000
```

## **Bit field**

In C, you can state the size of your structure (struct) or union members in the form of bits. This concept is to because of efficiently utilizing the memory when you know that your amount of a field or collection of fields is not going to exceed a specific limit or is in-between the desired range.

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows −

```
struct {
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows −

```
struct {
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept –

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

/* define simple structure */
struct {
   unsigned int widthValidated;
   unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
   unsigned int widthValidated : 1;
   unsigned int heightValidated : 1;
} status2;

int main( ) {
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
   getch();
   return 0;
}
```

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

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

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

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

The following table describes the variable elements of a bit field −

| Sr.No. | Element & Description |
|---|---|
| 1 | **Type** |

|  |  |
|---|---|
|  | An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int. |
| 2 | **member_name**<br><br>The name of the bit-field. |
| 3 | **Width**<br><br>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. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example

```
#include <stdio.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 );
  getch();
  return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result −

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

A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```
#include <stdio.h>

// A structure without forced alignment
struct test1 {
   unsigned int x : 5;
   unsigned int y : 8;
};

// A structure with forced alignment
struct test2 {
   unsigned int x : 5;
   unsigned int : 0;
   unsigned int y : 8;
};


int main()
{
   printf("Size of test1 is %lu bytes\n",
          sizeof(struct test1));
   printf("Size of test2 is %lu bytes\n",
          sizeof(struct test2));
   getch();
   return 0;
}
```

**Output:**

```
Size of test1 is 4 bytes

Size of test2 is 8 bytes
```

**More example of bit field**

Since we know that the value of date is always from 1 to 31, the value of month is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>

// Space optimized representation of the date
struct date {
    // d has value between 1 and 31, so 5 bits
    // are sufficient
    unsigned int d : 5;

    // m has value between 1 and 12, so 4 bits
    // are sufficient
    unsigned int m : 4;

    unsigned int y;
};

int main()
{
    printf("Size of date is %lu bytes\n", sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}
```

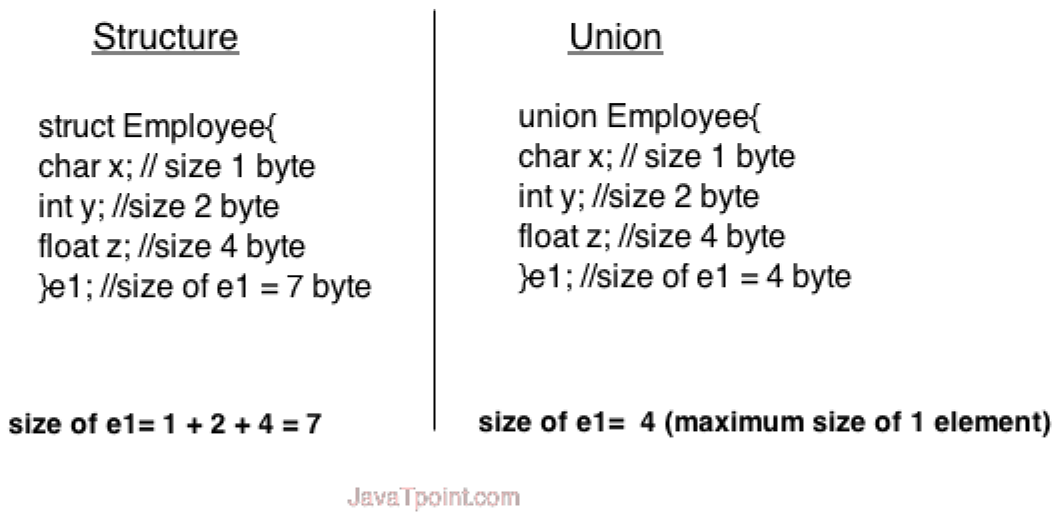
**Output:**

```
Size of date is 8 bytes

Date is 31/12/2014
```

## Union

Like structure, **Union in c language** is *a user-defined data type* that is used to store the different type of elements.

At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.


Structure
Union
size of e1= 1 + 2 + 4 = 7
size of e1= 4 (maximum size of 1 element)
JavaTpoint.com

```
struct Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 7 byte
```

```
union Employee{
char x; // size 1 byte
int y; //size 2 byte
float z; //size 4 byte
}e1; //size of e1 = 4 byte
```

## Advantage of union over structure

It **occupies less memory** because it occupies the size of the largest member only.

## Disadvantage of union over structure

Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

---

## Defining union

The **union** keyword is used to define the union. Let's see the syntax to define union in c.

```
union union_name
{
   data_type member1;
   data_type member2;
   .
   .
   data_type memeberN;
};
```

Let's see the example to define union for an employee in c.

```
union employee
{   int id;
```

```
    char name[50];
    float salary;
};
```

*C Union example*

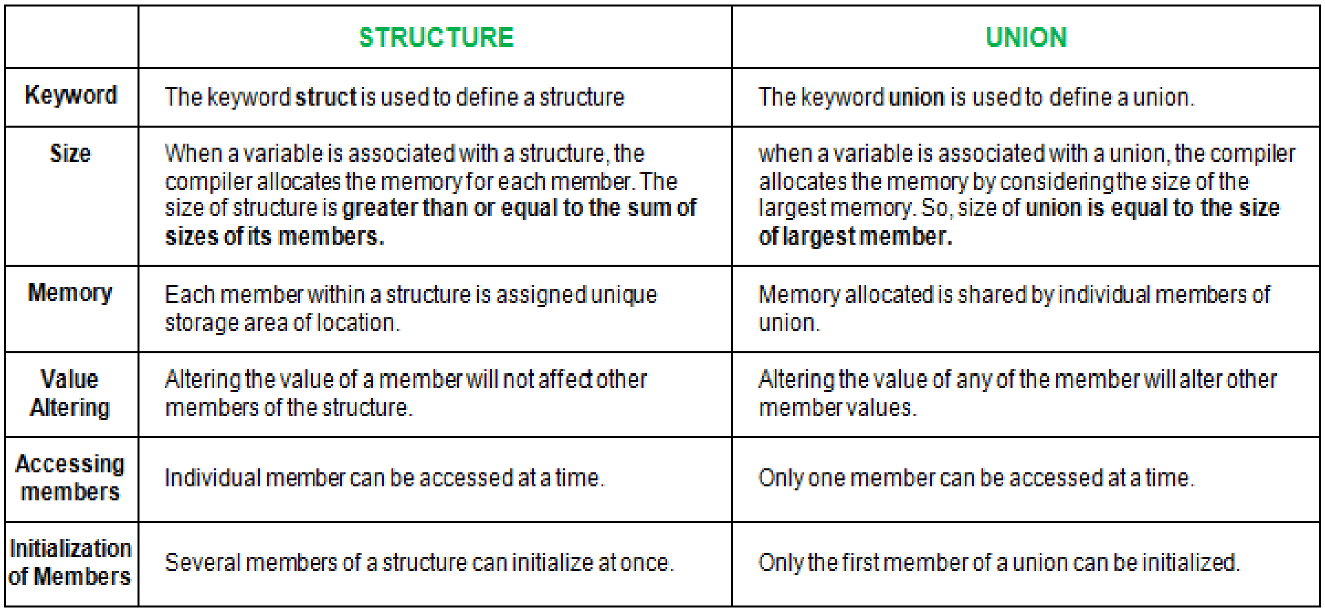
Let's see a simple example of union in C language.

```
#include <stdio.h>
#include <string.h>
union employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for union
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
   return 0;
}
```

Output:

```
employee 1 id : 1869508435
employee 1 name : Sonoo Jaiswal
```

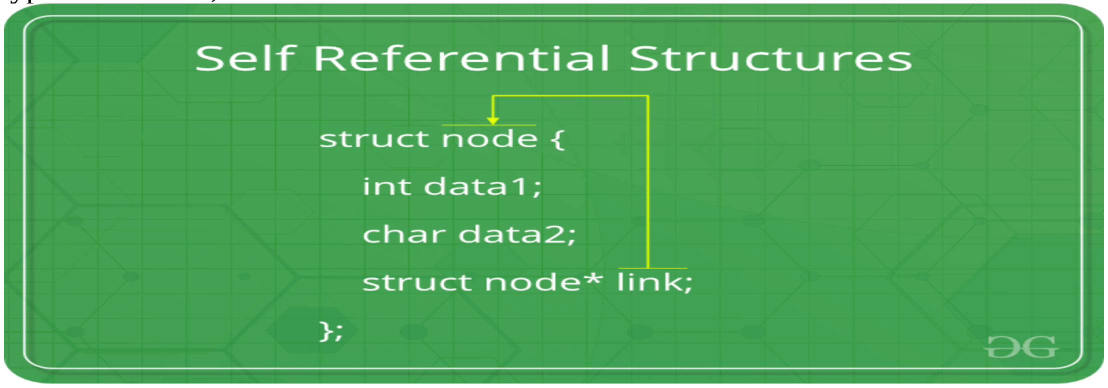
As you can see, id gets garbage value because name has large memory size. So only name will have actual value.



| | STRUCTURE | UNION |
|---|---|---|
| **Keyword** | The keyword **struct** is used to define a structure | The keyword **union** is used to define a union. |
| **Size** | When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.** | when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.** |
| **Memory** | Each member within a structure is assigned unique storage area of location. | Memory allocated is shared by individual members of union. |
| **Value Altering** | Altering the value of a member will not affect other members of the structure. | Altering the value of any of the member will alter other member values. |
| **Accessing members** | Individual member can be accessed at a time. | Only one member can be accessed at a time. |
| **Initialization of Members** | Several members of a structure can initialize at once. | Only the first member of a union can be initialized. |

## Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

In other words, structures pointing to the same type of structures are self-referential in nature.

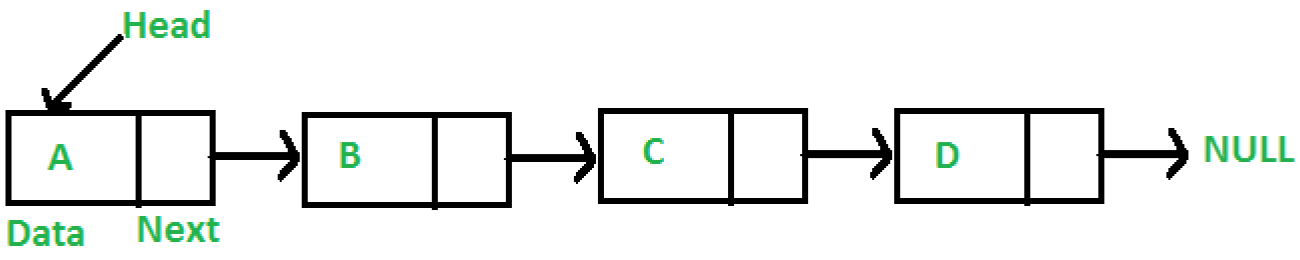
Example:

```
struct node {
   int data1;
   char data2;
   struct node* link;
};

int main()
{
   struct node ob;
   return 0;
```

}
In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

## Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

```
// A linked list node
struct Node {
   int data;
   struct Node* next;
};
```

### Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.
1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

## Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

## Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

## Types of Linked Lists

- singly linked list
- doubly linked list
- Circular linked list

## **File handling**

**FILE**

File is a collection of bytes that is stored on secondary storage devices like disk. There are two kinds of files in a system. They are,

1. Text files (ASCII)
2. Binary files

- Text files contain ASCII codes of digits, alphabetic and symbols.
- Binary file contains collection of bytes (0's and 1's). Binary files are compiled version of text files.

**Why files are needed?**

- 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.

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

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

---

## Functions for file handling

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

| No. | Function | Description |
|---|---|---|
| 1 | fopen() | opens new or existing file |
| 2 | fprintf() | write data into the file |
| 3 | fscanf() | reads data from the file |
| 4 | putc() | writes a character into the file |
| 5 | getc() | reads a character from file |
| 6 | fclose() | closes the file |
| 7 | fseek() | sets the file pointer to given position |
| 8 | putw() | writes an integer to file |
| 9 | getw() | reads an integer from file |
| 10 | ftell() | returns current position |
| 11 | rewind() | sets the file pointer to the beginning of the file |

## Opening File: fopen()

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

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

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

FILE is key word.

The fopen() function accepts two parameters:

- The file name (string).
- The mode in which the file is to be opened. It is a string.

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

| Mode | Description |
|---|---|
| r | Opens a file in read mode and sets pointer to the first character in the file. It returns null if file does not exist. |
| w | Opens a file in write mode. It returns null if file could not be opened. If file exists, data are overwritten. |
| a | Opens a file in append mode. If file is exist with data then contents of file will not be modified and new data will be added at the end of current data. It returns null if file couldn't be opened. |
| r+ | Opens a file for read and write mode and sets pointer to the first character in the file. |
| w+ | Opens a file for read and write mode and sets pointer to the first character in the file. |
| a+ | Opens a file for read and write mode and sets pointer to the first character in the file. But, it can't modify existing contents. |

The fopen function works in the following way.

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

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

```
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp = fopen("one.txt", "w");
   printf("Enter data...");
   while( (ch = getchar()) != EOF) {
      putc(ch, fp);
   }
   fclose(fp);
   fp = fopen("one.txt", "r");

   while( (ch = getc(fp))!= EOF)
   printf("%c",ch);

   // closing the file pointer
   fclose(fp);
   getch();

   return 0;
}
```

*Output*

The content of the file will be printed.

## Closing File: fclose()

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

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

Example

```
fclose(fp);
```

## Writing File : fprintf() function

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

**Syntax:**

int fprintf(FILE *fp, const char *format, …)
 we use fprinf() as below.
fprintf (fp, “%s %d”, “var1”, var2);

Where, fp is file pointer to the data type “FILE”.
var1 – string variable
var2 – Integer variable
This is for example only. You can use any specifies with any data type as we use in normal printf() function.

**Example:**

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

## Reading File : fscanf() function

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

```
int fscanf(FILE *fp, const char *format, …)
```

we use fscanf() as below.

```
fscanf (fp, “%d”, &age);
```

Where, fp is file pointer to the data type “FILE”.
age – Integer variable

**Example:**

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

Output:

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

WAP to create a file to store some data into the file and print the content of the file

```
#include <stdio.h>
void main()
{
  FILE *fp;
  char buff[255];//creating char array to store data of file

  fp = fopen("file.txt", "w");//opening file
```

```
   fprintf(fp, "Hello file by fprintf...\n");//writing data into file
   fclose(fp);//closing file
   fp = fopen("file.txt", "r");
   while(fscanf(fp, "%s", buff)!=EOF){
   printf("%s ", buff );
   }
   fclose(fp);
 getch();
}
```

Output

```
Hello file by fprintf
```

## Writing File : putc() function

putc() function is used to write a character into a file. It writes a single character at a time in a file and moves the file pointer position to the next address/location to write the next character.

Declaration:

```
 int putc(int char, FILE *fp)
```

we use fputc() function as below.

**putc**(ch, fp);

where,

ch – character value

fp – file pointer

**Example:**

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

**file1.txt**

```
a
```

## Reading File : getc() function

getc() is used to read a character from a file. It reads single character at a time and moves the file pointer position to the next address/location to read the next character.

Declaration:
 int **getc**(FILE *fp)
we use getc() function as below.
**getc** (fp);
where,
fp – file pointer

**Example:**

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

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

**myfile.txt**

```
this is simple text message
```

WAP to create a file to store some data into the file and print the content of the file

```
#include<stdio.h>


int main()
{
   FILE *fp;
```

```
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data...");
    while( (ch = getchar()) != EOF) {
        putc(ch, fp);
    }
    fclose(fp);
    fp = fopen("one.txt", "r");

    while( (ch = getc(fp))!= EOF)
    printf("%c",ch);

    // closing the file pointer
    fclose(fp);
    getch();
    return 0;
}
```

*Output*

The content of the file will be printed.

## Writing File : fputs() function

fputs() function is used to write a line of characters to the output screen.

Declaration:
int f**puts**(const char *string, FILE *fp)
we use puts function as below.
**fputs**(string,fp);

**Example:**

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
clrscr();
fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);

fclose(fp);
getch();
}
```

**myfile2.txt**

```
hello c programming
```

## Reading File : fgets() function

fgets() function is a file handling function in C programming language which is used to read a file line by line

Declaration:
char ***fgets**(char *string, int n, FILE *fp)
we use fgets function as below.

**fgets** (buffer, size, fp);
where
buffer – buffer to  put the data in.

size – size of the buffer

fp – file pointer

**Example:**

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
clrscr();

fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));
fclose(fp);
getch();
}
```

Output:

```
hello c programming
```

WAP to create a file to read a string into the file and print the content of the file

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char text[300];
fp=fopen("myfile2.txt","w");
fputs("hello c programming",fp);
fclose(fp);
fp=fopen("myfile2.txt","r");
printf("%s",fgets(text,200,fp));
fclose(fp);
getch();
}
```

Output:

```
hello c programming
```

## getw() and putw()

putw(), getw() functions are file handling function in C programming language which is used to write an integer value into a file (putw) and read integer value from a file (getw).

```
Declaration:
 int putw(int number, FILE *fp);
where
i – integer value
fp – file pointer

Declaration:
 int getw(FILE *fp);
```

getw function reads an integer value from a file pointed by fp. In a C program, we can read integer value from a file as below.
getw(fp);

```
#include <stdio.h>
int main ()
{

   FILE *fp;
   int i=1, j=2, k=3, num;
   fp = fopen ("test.c","w");
   putw(i,fp);
   putw(j,fp);
   putw(k,fp);
   fclose(fp);
   fp = fopen ("test.c","r");


   while((num=getw(fp))!=EOF)
      printf(“Data in test.c file is %d \n”, num);
   fclose(fp);
   getch();
   return 0;
}
```

***OUTPUT:***

```
Data in test.c file is
1
2
3
```

## fseek()

fseek() functions is used to put file pointer at desired location..

It has following constants. SEEK_SET, SEEK_CUR, SEEK_END.

Declaration:
 int fseek(FILE *fp, long int offset, int position)
where,
fp – file pointer
offset – Number of bytes/characters to be offset/moved from position/the current file pointer position

position – This is the current file pointer position from where offset is added. Below 3 constants are used to specify this.

SEEK_SET – It moves file pointer position to the beginning of the file.

SEEK_CUR – It moves file pointer position to current location.

SEEK_END – It moves file pointer position to the end of file.

```
#include <stdio.h>
int main ()
{
  FILE *fp;
  char data[60];
  fp = fopen ("test.txt","w");
  fputs("Fresh2refresh.com is a programming tutorial website", fp);
  fclose(fp);
  fp = fopen ("test.txt","r");
  fgets ( data, 60, fp );
  printf("Before fseek - %s", data);

  // To set file pointet to 21th byte/character in the file
  fseek(fp, 21, SEEK_SET);
  fgets ( data, 60, fp );
  printf("\nAfter SEEK_SET to 21 - %s", data);

  // To find backward 10 bytes from current position
  fseek(fp, -10, SEEK_CUR);
  fgets ( data, 60, fp );
  printf("\nAfter SEEK_CUR to -10 - %s", data);

  // To find 7th byte before the end of file
  fseek(fp, -7, SEEK_END);
  fgets ( data, 60, fp );
  printf("\nAfter SEEK_END to -7 - %s", data);

  // To set file pointer to the beginning of the file
  fseek(fp, 0, SEEK_SET); // We can use rewind(fp); also

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

***OUTPUT:***

```
Before fseek – Fresh2refresh.com is a programming tutorial website
After SEEK_SET to 21 – a programming tutorial website
After SEEK_CUR to -10 – al website
After SEEK_END to -7 – website
```

## rewind()

rewind () function moves file pointer position to the beginning of the file.

Declaration:

```
 void rewind(FILE *fp)
```

we use rewind() as below.

```
rewind(fp);
```

both rewind()  and fseek(fp, 0, SEEK_SET) moves file pointer position to the beginning of the file.

## ftell()

ftell () function gives current position of file pointer.

Declaration:

```
 long int ftell(FILE *fp)
```

we use ftell() as below.

**ftell**(fp);

```
#include <stdio.h>
#include<conio.h>
void main () {
   FILE *f;
   int len;
   f = fopen("one.txt", "w");
   putc("surendra",f);
   fclose(f);
   f = fopen("one.txt", "r");
   fseek(f, 0, SEEK_END);
   len = ftell(f);
   fclose(f);
   printf("Size of file: %d bytes", len);
   getch();
}
```

**Output**

```
Size of file: 8  bytes.
```

## feof ()

feof () function finds end of file.

Declaration:
 int feof(FILE *fp)
 we use feof() function as below.
feof(fp);

where,
fp – file pointer

```
/* Open, Read and close a file: Reading char by char */

# include <stdio.h>
int main( )
{
   FILE *fp ;
   char c ;
   fp = fopen ( "test.c", "w" ) ;
   fputs("good morning",fp);
   fclose(fp);
      fp = fopen ( "test.c", "r" ) ; // opening an existing file
   if ( fp == NULL )
   {
     printf ( "Could not open file test.c" ) ;
     return 1;
   }
      while ( 1 )
   {
     c = getc ( fp ) ; // reading the file
     if( feof(fp) )
     break ;
     printf ( "%c", c ) ;
   }
   fclose ( fp ) ; // Closing the file
   getch();
   return 0;
}
```

**OUTPUT:**

```
good morning
```

### ferror()

This function tests the error of file pointer.

Following is the declaration for ferror() function.

```
int ferror(FILE *stream)
```

Parameters

- **stream** − This is the pointer to a FILE object that identifies the stream.

Return Value

If the error occures, the function returns a non-zero value else, it returns a zero value.

Example

The following example shows the usage of ferror() function.

```
#include <stdio.h>

int main () {
   FILE *fp;
   char c;
   fp = fopen("file.txt", "w");
   c = getc(fp);
   if( ferror(fp) ) {
      printf("Error in reading from file : file.txt\n");
   }
   fclose(fp);
   getch();
   return(0);
}
```

Assuming we have a text file **file.txt**, which is an empty file. Let us compile and run the above program that will produce the following result because we try to read a file which we opened in **write only** mode.

```
Error reading from file "file.txt"
```

## The fwrite() function

The fwrite() function is used to write records (sequence of bytes) to the file. A record may be an array or a structure.

**Syntax of fwrite() function**

```
        fwrite( ptr, int size, int n, FILE *fp );
```

The fwrite() function takes four arguments.
ptr : ptr is the reference of an array or a structure stored in memory.
size : size is the total number of bytes to be written.
n : n is number of times a record will be written.
fp is a file pointer where the records will be written in binary mode.

**Example of fwrite() function**

```
    #include<stdio.h>

    struct Student
    {
        int roll;
        char name[25];
        float marks;
    };

    void main()
    {
        FILE *fp;
        char ch;
        struct Student Stu;

        fp = fopen("Student.dat","w");          //Statement   1

        if(fp == NULL)
        {
            printf("\nCan't open file or file doesn't exist.");
            exit(0);
        }

        do
        {
             printf("\nEnter Roll : ");
             scanf("%d",&Stu.roll);

             printf("Enter Name : ");
             scanf("%s",Stu.name);

             printf("Enter Marks : ");
             scanf("%f",&Stu.marks);

             fwrite(&Stu,sizeof(Stu),1,fp);

             printf("\nDo you want to add another data (y/n) : ");
             ch = getche();

        }while(ch=='y' || ch=='Y');

        printf("\nData written successfully...");

        fclose(fp);
        getch();
    }
```

```
Output :

        Enter Roll : 1
        Enter Name : Ashish
        Enter Marks : 78.53

        Do you want to add another data (y/n) : y

        Enter Roll : 2
        Enter Name : Kaushal
        Enter Marks : 72.65

        Do you want to add another data (y/n) : y
        Enter Roll : 3
        Enter Name : Vishwas
        Enter Marks : 82.65

        Do you want to add another data (y/n) : n

        Data written successfully...
```

## The fread() function

The fread() function is used to read bytes form the file.

**Syntax of fread() function**

```
        fread( ptr, int size, int n, FILE *fp );
```

The fread() function takes four arguments.
ptr : ptr is the reference of an array or a structure where data will be stored after reading.
size : size is the total number of bytes to be read from file.
n : n is number of times a record will be read.
FILE* : FILE* is a file where the records will be read.

**Example of fread() function**

```
#include<stdio.h>

struct Student
{
    int roll;
    char name[25];
    float marks;
};

void main()
{
    FILE *fp;
    char ch;
    struct Student Stu;

    fp = fopen("Student.dat","r");          //Statement   1

    if(fp == NULL)
    {
        printf("\nCan't open file or file doesn't exist.");
        exit(0);
        }

        printf("\n\tRoll\tName\tMarks\n");

        while(fread(&Stu,sizeof(Stu),1,fp)>0)
          printf("\n\t%d\t%s\t%f",Stu.roll,Stu.name,Stu.marks);
        fclose(fp);
        getch();
    }
Output :

        Roll    Name     Marks
        1       Ashish   78.53
        2       Kaushal  72.65
        3       Vishwas   82.65
```

## Input and Output function

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

### printf() Function

The **printf** function reads the input from the standard input device and scans that input according to the **format** provided.

The prototype of function is **int printf (const char *format, ...)**

### scanf() function

reads the input from the standard input device and scans that input according to the **format** provided.

The prototype of function is **int scanf(const char *format, ...)**

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements.

```
#include <stdio.h>
int main( ) {
   char str[100];
   int i;
   printf( "Enter a value :");
   scanf("%s %d", str, &i);
   printf( "\nYou entered: %s %d ", str, i);
   getch();
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows

**Enter a value:** seven 7
**You entered:** seven 7

## The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

example −

```
#include <stdio.h>
int main( )
{

   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );
   getch();

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

**Enter a value :** this is test
**You entered:** t

## The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string 's' and a trailing newline to **stdout**.

```
#include <stdio.h>
int main( ) {
   char str[100];
   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );
   getch();
   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

**Enter a value :** this is test
**You entered:** this is test

## Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

| Format specifier | Description | Supported data types |
|---|---|---|
| %c | Character | char<br>unsigned char |
| %d | Signed Integer | short<br>unsigned short<br>int<br>long |
| %e or %E | Scientific notation of float values | float<br>double |
| %f | Floating point | float |
| %g or %G | Similar as %e or %E | float<br>double |
| %hi | Signed Integer(Short) | short |
| %hu | Unsigned Integer(Short) | unsigned short |
| %i | Signed Integer | short<br>unsigned short<br>int<br>long |
| %l or %ld or %li | Signed Integer | long |
| %lf | Floating point | double |

| Format specifier | Description | Supported data types |
|---|---|---|
| %Lf | Floating point | long double |
| %lu | Unsigned integer | unsigned int<br>unsigned long |
| %lli, %lld | Signed Integer | long long |
| %llu | Unsigned Integer | unsigned long long |
| %o | Octal representation of Integer. | short<br>unsigned short<br>int<br>unsigned int<br>long |
| %p | Address of pointer to void void * | void * |
| %s | String | char * |
| %u | Unsigned Integer | unsigned int<br>unsigned long |
| %x or %X | Hexadecimal representation of Unsigned Integer | short<br>unsigned short<br>int<br>unsigned int<br>long |

| **Format specifier** | **Description** | **Supported data types** |
|---|---|---|
| %n | Prints nothing | |
| %% | Prints % character | |