UNIT –II

Arrays in C

When we work with large number of data values we need that many number of different variables. As the number of variables increases, the complexity of the program also increases and so the programmers get confused with the variable names. There may be situations where we need to work with large number of similar data values. To make this work more easy, C programming language provides a concept called "Array".

An array is a special type of variable used to store multiple values of same data type at a time.

An array can also be defined as follows...

An array is a collection of similar data items stored in continuous memory locations with single name.

Declaration of an Array

In c programming language, when we want to create an array we must know the datatype of values to be stored in that array and also the number of values to be stored in that array.

We use the following general syntax to create an array...

datatypearrayName [size] ;

Syntax for creating an array with size and initial values

datatypearrayName [size] = {value1, value2, ...} ;

Syntax for creating an array without size and with initial values

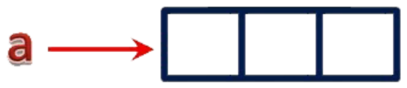
datatypearrayName [ ] = {value1, value2, ...} ;

In the above syntax, the **datatype** specifies the type of values we store in that array and **size** specifies the maximum number of values that can be stored in that array.

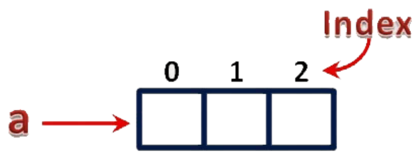
Example

int a [3] ;

Here, the compiler allocates 6 bytes of continuous memory locations with single name 'a' and tells the compiler to store three different integer values (each in 2 bytes of memory) into that 6 bytes of memory. For the above declaration the memory is organized as follows...



In the above memory allocation, all the three memory locations has a common name 'a'. So the accession of individual memory location is not possible directly. Hence compiler not only allocates the memory but also assigns a numerical reference value to every individual memory location of an array. This reference number is called as "Index" or "subscript" or "indices". Index values for the above example is as follows...

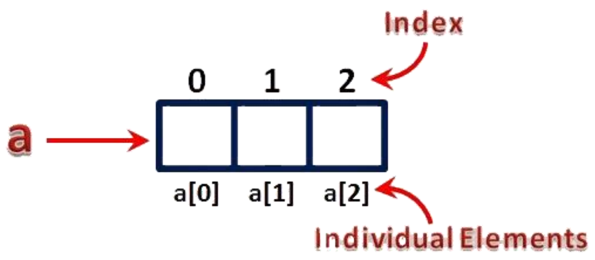


Accessing Individual Elements of an Array

The individual elements of an array are identified using the combination of 'arrayName' and 'indexValue'. We use the following general syntax to access individual elements of an array...

arrayName [indexValue] ;

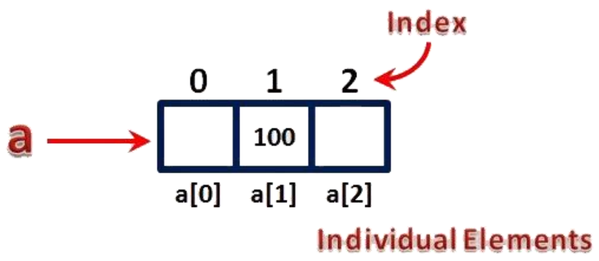
For the above example the individual elements can be denoted as follows...



For example, if we want to assign a value to the second memory location of above array 'a', we use the following statement...

a [1] = 100 ;

The result of above assignment statement is as follows...



Types of Arrays in C

In c programming language, arrays are classified into **two types**. They are as follows...

1. **Single Dimensional Array / One Dimensional Array**
2. **Multi Dimensional Array**

Single Dimensional Array

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as **one-dimensional arrays**, **Linear Arrays** or simply **1-D Arrays**.

Declaration of Single Dimensional Array

We use the following general syntax for declaring a single dimensional array...

datatypearrayName [size] ;

Example

introllNumbers [60] ;

The above declaration of single dimensional array reserves 60 continuous memory locations of 2 bytes each with the name **rollNumbers** and tells the compiler to allow only integer values into those memory locations.

Initialization of Single Dimensional Array

We use the following general syntax for declaring and initializing a single dimensional array with size and initial values.

datatypearrayName [size] = {value1, value2, ...} ;

Example

int marks [6] = { 89, 90, 76, 78, 98, 86 } ;

The above declaration of single dimensional array reserves 6 continuous memory locations of 2 bytes each with the name **marks** and initializes with value 89 in first memory location, 90 in second memory location, 76 in third memory location, 78 in fourth memory location, 98 in fifth memory location and 86 in sixth memory location.

We can also use the following general syntax to intialize a single dimensional array without specifying size and with initial values...

datatypearrayName [ ] = {value1, value2, ...} ;

The array must be initialized if it is created without specifying any size. In this case, the size of the array is decided based on the number of values initialized.

Example

int marks [] = { 89, 90, 76, 78, 98, 86 } ;

charstudentName [] = "btechsmartclass" ;

In the above example declaration, size of the array **'marks'** is **6** and the size of the array **'studentName'** is **16**. This is because in case of character array, compiler stores one exttra character called **0** (NULL) at the end.

Accessing Elements of Single Dimensional Array

In c programming language, to access the elements of single dimensional array we use array name followed by index value of the element that to be accessed. Here the index value must be enclosed in square braces. **Index** value of an element in an array is the reference number given to each element at the time of memory allocation. The index value of single dimensional array starts with zero (0) for first element and incremented by one for each element. The index value in an array is also called as **subscript** or **indices**.

We use the following general syntax to access individual elements of single dimensional array...

arrayName [indexValue]

Example

marks [2] = 99 ;

In the above statement, the third element of **'marks'** array is assinged with value **'99'**.

Multi Dimensional Array

An array of arrays is called as multi dimensional array. In simple words, an array created with more than one dimension (size) is called as multi dimensional array. Multi dimensional array can be of **two dimensional array** or **three dimensional array** or **four dimensional array** or more...

Most popular and commonly used multi dimensional array is **two dimensional array**. The 2-D arrays are used to store data in the form of table. We also use 2-D arrays to create mathematical **matrices**.

Declaration of Two Dimensional Array

We use the following general syntax for declaring a two dimensional array...

datatypearrayName [rowSize] [columnSize] ;

Example

intmatrix_A [2][3] ;

The above declaration of two dimensional array reserves 6 continuous memory locations of 2 bytes each in the form of **2 rows** and **3 columns**.

Initialization of Two Dimensional Array

We use the following general syntax for declaring and initializing a two dimensional array with specific number of rows and coloumns with initial values.

datatypearrayName [rows][colmns] = {{r1c1value, r1c2value, ...},{r2c1, r2c2,...}...} ;

Example

intmatrix_A [2][3] = { {1, 2, 3},{4, 5, 6} } ;

The above declaration of two dimensional array reserves 6 continuous memory locations of 2 bytes each in the form of 2 rows and 3 coloumns. And the first row is initialized with values 1, 2 & 3 and second row is initialized with values 4, 5 & 6.

We can also initialize as follows...

```
intmatrix_A [2][3] = {
                  {1, 2, 3},
                  {4, 5, 6}
                } ;
```

Accessing Individual Elements of Two Dimensional Array

In c programming language, to access elements of a two dimensional array we use array name followed by row index value and column index value of the element that to be accessed. Here the row and column index values must be enclosed in separate square braces. In case of two dimensional array the compiler assigns separate index values for rows and columns.

We use the following general syntax to access the individual elements of a two dimensional array...

arrayName [rowIndex] [columnIndex]

Example

matrix_A [0][1] = 10 ;

In the above statement, the element with row index 0 and column index 1 of **matrix_A** array is assinged with value **10**.

Applications of Arrays in C

In c programming language, arrays are used in wide range of applications. Few of them are as follows...

Arrays are used to Store List of values

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

Arrays are used to Perform Matrix Operations

We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.

Arrays are used to implement Search Algorithms

We use single dimensional arrays to implement search algorihtmslike ...

1. Linear Search
2. Binary Search

Arrays are used to implement Sorting Algorithms

We use single dimensional arrays to implement sorting algorihtmslike ...

1. Insertion Sort
2. Bubble Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort, etc.,

Arrays are used to implement Datastructures

We use single dimensional arrays to implement datastructures like...

1. Stack Using Arrays
2. Queue Using Arrays

Arrays are also used to implement CPU Scheduling Algorithms

Advantage of C Array

1) Code Optimization: Less code to the access the data.

2) Easy to traverse data: By using the for loop, we can retrieve the elements of an array easily.

3) Easy to sort data: To sort the elements of array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

1) Fixed Size: Whatever size, we define at the time of declaration of array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

We can declare an array in the c language in the following way.

```
data_type array_name[array_size];
```

Now, let us see the example to declare array.

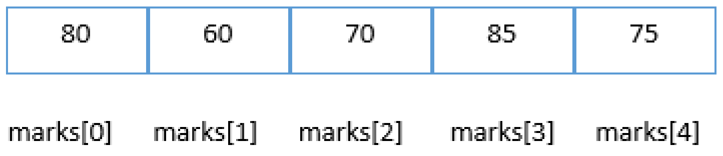
```
int marks[5];
```

Here, int is the *data_type*, marks is the *array_name* and 5 is the *array_size*.

Initialization of C Array

A simple way to initialize array is by index. Notice that **array index starts from 0** and ends with [SIZE - 1].

```
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```



Initialization of Array

C array example

```
#include <stdio.h>
#include <conio.h>
void main(){
int i=0;
int marks[5];//declaration of array
clrscr();

marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;

//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}//end of for loop

getch();
}
```

Output

```
80
60
70
85
75
```

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

```
int marks[5]={20,30,40,50,60};
```

In such case, there is **no requirement to define size**. So it can also be written as the following code.

```
int marks[]={20,30,40,50,60};
```

Let's see the full program to declare and initialize the array in C.

```
#include <stdio.h>
#include <conio.h>
void main(){
int i=0;
int marks[5]={20,30,40,50,60};//declaration and initialization of array
clrscr();

//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}

getch();
}
```

Output

```
20
30
40
50
60
```

Two Dimensional Array in C

The two dimensional array in C language is represented in the form of rows and columns, also known as matrix. It is also known as *array of arrays* or *list of arrays*.

The two dimensional, three dimensional or other dimensional arrays are also known as *multidimensional* arrays.

Declaration of two dimensional Array in C

We can declare an array in the c language in the following way.

data_type array_name[size1][size2];

A simple example to declare two dimensional array is given below.

int twodimen[4][3];

Here, 4 is the *row* number and 3 is the *column* number.

Initialization of 2D Array in C

A way to initialize the two dimensional array at the time of declaration is given below.

int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};

Two dimensional array example in C

```
#include <stdio.h>
#include <conio.h>
void main(){
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
clrscr();

//traversing 2D array
for(i=0;i<4;i++){
 for(j=0;j<3;j++){
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i

getch();
}
```

Output

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
```

```
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

C Strings

String in C language is an *array of characters* that is terminated by \0 (null character).

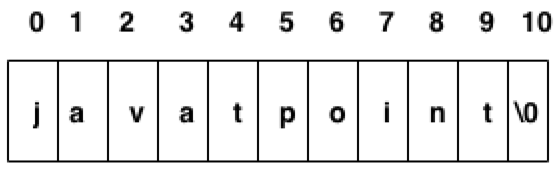
There are two ways to declare string in c language.

1. By char array
2. By string literal

Let's see the example of declaring **string by char array** in C language.

char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

As you know well, array index starts from 0, so it will be represented as in the figure given below.



While declaring string, size is not mandatory. So you can write the above code as given below:

char ch[]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};

You can also define **string by string literal** in C language. For example:

char ch[]="javatpoint";

In such case, '\0' will be appended at the end of string by the compiler.

Difference between char array and string literal

The only difference is that string literal cannot be changed whereas string declared by char array can be changed.

String Example in C

Let's see a simple example to declare and print string. The '%s' is used to print string in c language.

```
#include <stdio.h>
void main ()
{
   char ch[11]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
   char ch2[11]="javatpoint";

   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
}
```

Output:

```
Char Array Value is: javatpoint
String Literal Value is: javatpoint
```

C gets() and puts() functions

The gets() function reads string from user and puts() function prints the string. Both functions are defined in <stdio.h> header file.

Let's see a simple program to read and write string using gets() and puts() functions.

```
#include<stdio.h>
#include<conio.h>
void main(){
    char name[50];
    clrscr();
    printf("Enter your name: ");
    gets(name); //reads string from user
    printf("Your name is: ");
    puts(name);  //displays string
    getch();
}
```

Output:

```
Enter your name: Sonoo Jaiswal
Your name is: Sonoo Jaiswal
```

C String Functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.

2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

C String Length: strlen() function

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
#include <stdio.h>
void main()
{
  char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
  printf("Length of string is: %d",strlen(ch));
}
```

Output:

```
Length of string is: 10
```

C Copy String: strcpy()

The strcpy(destination, source) function copies the source string in destination.

```
#include <stdio.h>
void main()
{
  char ch[20]={'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
  char ch2[20];
  strcpy(ch2,ch);
  printf("Value of second string is: %s",ch2);
}
```

Output:

```
Value of second string is: javatpoint
```

C String Concatenation: strcat()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

```
#include <stdio.h>
void main()
{
   char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
   char ch2[10]={'c', '\0'};
   strcat(ch,ch2);
   printf("Value of first string is: %s",ch);
}
```

Output:

```
Value of first string is: helloc
```

C Compare String: strcmp()

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

```
#include <stdio.h>
void main()
{
   char str1[20],str2[20];
  printf("Enter 1st string: ");
  gets(str1);//reads string from console
  printf("Enter 2nd string: ");
  gets(str2);
  if(strcmp(str1,str2)==0)
      printf("Strings are equal");
  else
      printf("Strings are not equal");
}
```

Output:

```
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal
```

C Reverse String: strrev()

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```
#include<stdio.h>
#include<conio.h>
void main(){
  char str[20];
  clrscr();
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nReverse String is: %s",strrev(str));
  getch();
}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Reverse String is: tnioptavaj
```

C String Lowercase: strlwr()

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
#include<stdio.h>
#include<conio.h>
void main(){
  char str[20];
  clrscr();
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nLower String is: %s",strlwr(str));
  getch();
}
```

Output:

```
Enter string: JAVATpoint
String is: JAVATpoint
Lower String is: javatpoint
```

C String Uppercase: strupr()

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

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

```
#include<conio.h>
void main(){
  char str[20];
  clrscr();
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nUpper String is: %s",strupr(str));
  getch();
}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

C String strstr()

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main(){
  char str[100]="this is javatpoint with c and java";
  char *sub;
  clrscr();
  sub=strstr(str,"java");
  printf("\nSubstring is: %s",sub);
  getch();
}
```

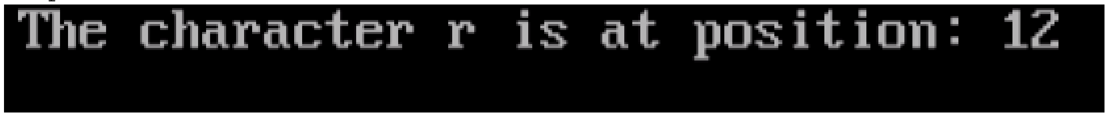
Output:

```
javatpoint with c and java
```

strchr() example:

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

void main(void)
{
char string[15];
char *ptr, c = 'r';
clrscr();
strcpy(string, "This is a string");
ptr = strrchr(string, c);
if (ptr)
printf("The character %c is at position: %d\n", c, ptr-string);
else
printf("The character was not found\n");
}
```

Output:



```
The character r is at position: 12
```

Array of Strings

Sorting names in alphabetic order:

```
#include<stdio.h>
#include<string.h>
int main(){
inti,j,count;
charstr[25][25],temp[25];
clrscr();
printf("How many strings u are going to enter?: ");
scanf("%d",&count);

puts("Enter Strings one by one: ");
for(i=0;i<count;i++)
scanf("%s",&str[i]);

for(i=0;i<count;i++)
for(j=0;j<count-1;j++){
            if(strcmp(str[j],str[j+1])>0){
            strcpy(temp,str[j]);
            strcpy(str[j],str[j+1]);
            strcpy(str[j+1],temp);
             }
    }
printf("Order of Sorted Strings:");
for(i=0;i<count;i++)
printf("%s ",str[i]);
getch();
return 0;
}
```

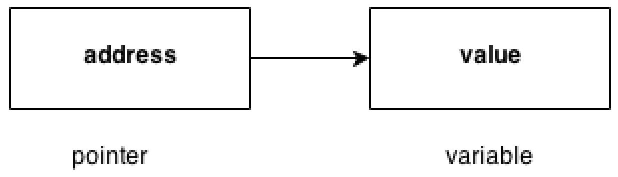
Program to find entered name exist in the list or not.

```
#include<stdio.h>
#incluDe<conio.h>
void main( )
{
charmasterlist[6][10] = {
"akshay",
"parag",
"raman",
"srinivas",
"gopal",
"rajesh"
} ;
inti, flag, a ;
charyourname[10] ;
printf ( "\nEnter your name " ) ;
scanf ( "%s", yourname ) ;
flag = 0 ;
for ( i = 0 ; i<= 5 ; i++ )
{
a = strcmp( &masterlist[i][0], yourname ) ;
if ( a == 0 )
{
printf ( "name found" ) ;
flag = 1 ;
break ;
}
}
if ( flag ==0 )
printf ( "Name not found" ) ;
}
```

Function	**Use**
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

C Pointers

The **pointer in C language** is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can **return multiple values from function** using pointer.

3) It makes you able to **access any memory location** in the computer's memory.

Usage of pointer

There are many usage of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	address of operator	determines the address of a variable.
* (asterisk sign)	indirection operator	accesses the value at the address.

Address Of Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

Pointer Notation

Consider the declaration,

```
inti = 3 ;
```

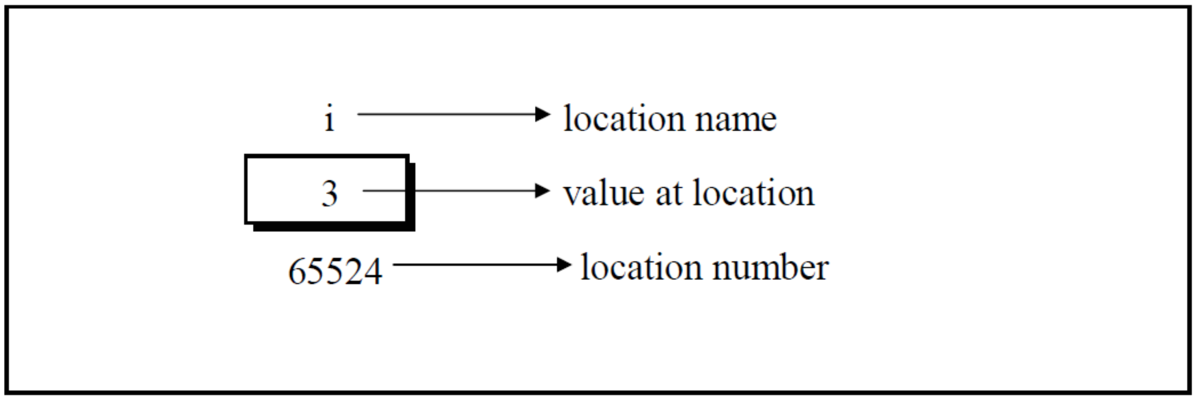
This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i**with this memory location.

(c) Store the value 3 at this location.

We may represent **i**'s location in memory by the following memory map.



```
voidmain( )
{
inti = 3 ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

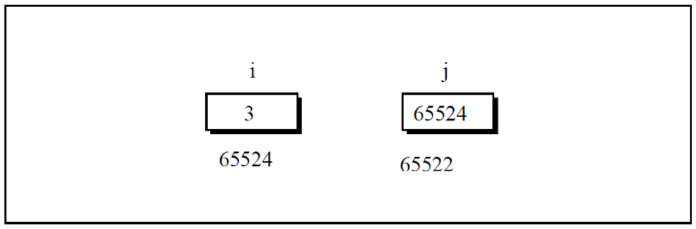
Note that printing the value of ***(&i)** is same as printing the valueof **i**.

The expression **&i**gives the address of the variable **i**. This addresscan be collected in a variable, by saying,

```
j = &i ;
```

But remember that **j** is not an ordinary variable like any otherinteger variable. It is a variable that contains the address of othervariable (**i**in this case). Since **j** is a variable the compiler mustprovide it space in the memory. Once again, the following memory

map would illustrate the contents of **i**and **j**.



```
voidmain( )
{
inti = 3 ;
int *j ;
j = &i ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nValue of j = %u", j ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3

Value of i = 3
```

```
#include <stdio.h>
#include <conio.h>
void main(){
int number=50;
clrscr();
printf("value of number is %d, address of number is %u",number,&number);
getch();
}
```

Output

```
value of number is 50, address of number is fff4
```

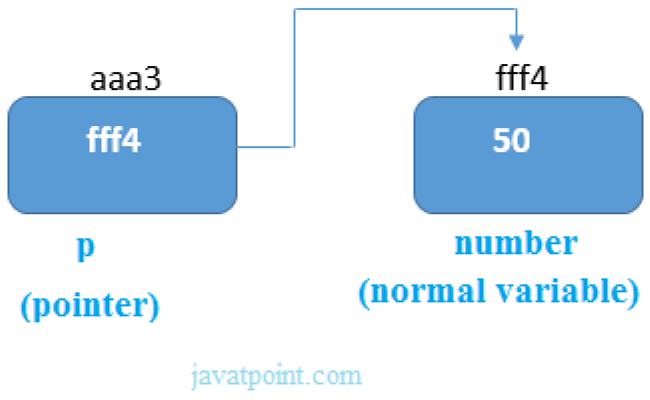
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol).

```
int *a;//pointer to int
char *c;//pointer to char
```

Pointer example

An example of using pointers printing the address and value is given below.



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.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

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

```
#include <stdio.h>
#include <conio.h>
void main(){
int number=50;
int *p;
clrscr();
p=&number;//stores the address of number variable

printf("Address of number variable is %x \n",&number);
printf("Address of p variable is %x \n",p);
printf("Value of p variable is %d \n",*p);

getch();
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

NULL Pointer

A pointer that is not assigned any value but NULL is known as NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will a better approach.

```
int *p=NULL;
```

In most the libraries, the value of pointer is 0 (zero).

Pointer Program to swap 2 numbers without using 3rd variable

```
#include<stdio.h>
#include<conio.h>
void main(){
int a=10,b=20,*p1=&a,*p2=&b;
clrscr();

printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);

getch();
}
```

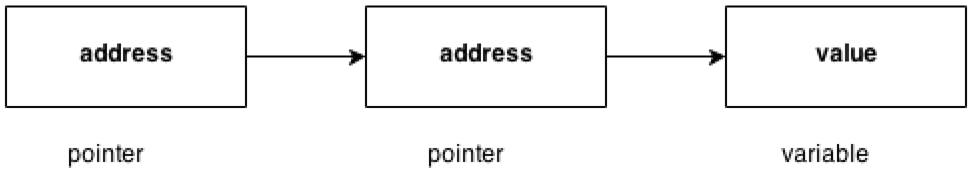
Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

C Pointer to Pointer

In C pointer to pointer concept, a pointer refers to the address of another pointer.

In c language, a pointer can point to the address of another pointer which points to the address of a value. Let's understand it by the diagram given below:



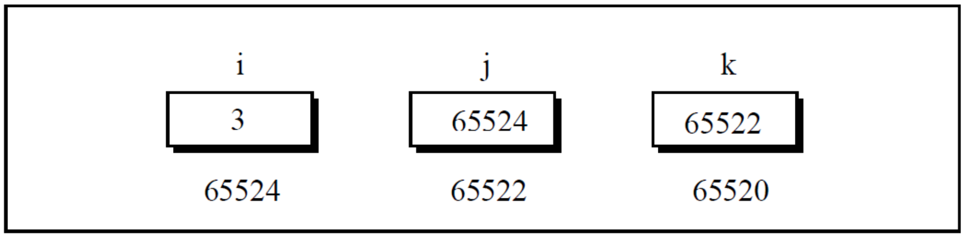
Let's see the syntax of pointer to pointer.

1. **int** **p2;

```
main( )
{
inti = 3, *j, **k ;
j = &i ;
k = &j ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of i = %u", *k ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nAddress of j = %u", k ) ;
printf ( "\nAddress of k = %u", &k ) ;
printf ( "\nValue of j = %u", j ) ;
printf ( "\nValue of k = %u", k ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", * ( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
printf ( "\nValue of i = %d", **k ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j = 65524
Value of k = 65522
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
```



Pointers to Pointers in C

In c programming language, we have pointers to store the address of variables of any datatype. A pointer variable can store the address of normal variable. C programming language also provides pointer variable to store the address of another pointer variable. This type of pointer variable is called as pointer to pointer variable. Sometimes we also call it as double pointer. We use the following syntax for creating pointer to pointer…

datatype **pointerName ;

Example Program

```
int **ptr ;
```

Here, **ptr** is an integer pointer variable that stores the address of another integer pointer variable but does not stores the normal integer variable address.

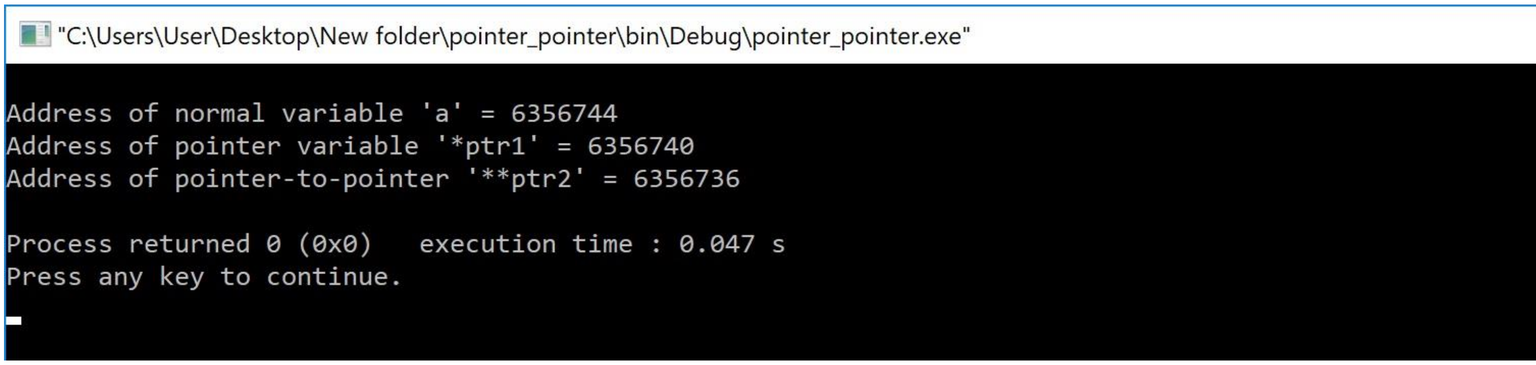
MOST IMPORTANT POINTS TO BE REMEMBERED

1. To store the address of normal variable we use single pointer variable
2. To store the address of single pointer variable we use double pointer variable
3. To store the address of double pointer variable we use triple pointer variable
4. Similarly the same for remaining pointer variables also…

Example Program

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

int main()
{
int a ;
int *ptr1 ;
int **ptr2 ;
int ***ptr3 ;

   ptr1 = &a ;
   ptr2 = &ptr1 ;
   ptr3 = &ptr2 ;

printf("\nAddress of normal variable 'a' = %u\n", ptr1) ;
printf("Address of pointer variable '*ptr1' = %u\n", ptr2) ;
printf("Address of pointer-to-pointer '**ptr2' = %u\n", ptr3) ;
return 0;
}
```

Output:


"C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe"

```
Address of normal variable 'a' = 6356744
Address of pointer variable '*ptr1' = 6356740
Address of pointer-to-pointer '**ptr2' = 6356736

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

Pointer Arithmetic in C

In C pointer holds address of a value, so there can be arithmetic operations on the pointer variable. Following arithmetic operations are possible on pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

Incrementing a pointer is used in array because it is contiguous memory location. Moreover, we know the value of next location.

Increment operation depends on the data type of the pointer variable. The formula of incrementing pointer is given below:

1. new_address= current_address + i * size_of(data type)

32 bit

For 32 bit int variable, it will increment to 2 byte.

64 bit

For 64 bit int variable, it will increment to 4 byte.

Let's see the example of incrementing 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 increment: Address of p variable is %u \n",p);
}
```

Output

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

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. The formula of decrementing pointer is given below:

1. new_address= current_address - i * size_of(data type)

32 bit

For 32 bit int variable, it will decrement to 2 byte.

64 bit

For 64 bit int variable, it will decrement to 4 byte.

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);
}
```

Output

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

C Pointer Addition

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

1. 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 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+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
}
```

Output

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

As you can see, 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 OS, it increments 12. But if we were using 32 bit OS, it were incrementing to 6 only i.e. 2*3=6. As integer value occupies 2 byte memory in 32 bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. The formula of subtracting value from pointer variable is given below:

1. 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 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-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
}
```

Output

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

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

Pointers to Arrays in C

In c programming language, when we declare an array the compiler allocates required amount of memory and also creates constant pointer with array name and stores the base address of that pointer in it. The address of first element of an array is called as **base address** of that array.

The array name itself acts as pointer to the first element of that array. Consider the following example of array declaration...

Example Code

```
int  marks[6] ;
```

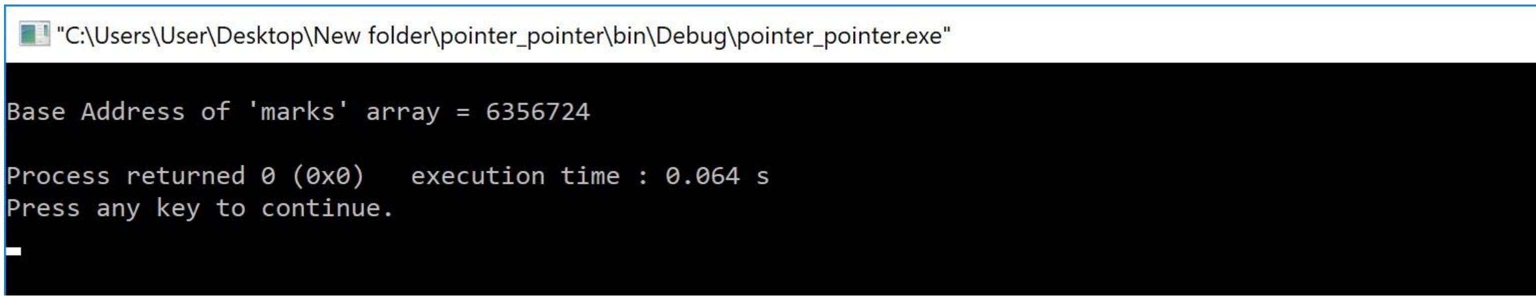
For the above declaration, the compiler allocates 12 bytes of memory and the address of first memory location (i.e., marks[0]) is stored in a constant pointer called **marks**. That means in the above example, **marks** is a pointer to **marks[0]**.

Example Program

```
#include<stdio.h>

#include<conio.h>
```

```
int main()

{

int marks[6] = {89, 45, 58, 72, 90, 93} ;

int *ptr ;


clrscr() ;


ptr = marks ;

printf(“Base Address of 'marks' array = %u\n”, ptr) ;

return 0;

}
```

Output:



"C:\Users\User\Desktop\New folder\pointer_pointer\bin\Debug\pointer_pointer.exe"

```
Base Address of 'marks' array = 6356724

Process returned 0 (0x0)   execution time : 0.064 s
Press any key to continue.
```

MOST IMPORTANT POINTS TO BE REMEMBERED

1. An array name is a **constant pointer**.
2. We can use the array name to access the address and value of all the elements of that array.
3. Since array name is a constant pointer we can't modify its value.

Consider the following example statements...

Example Code

```
ptr = marks + 2 ;
```

Here, the pointer variable **"ptr"** is assigned with address of **"marks[2]"** element.

Example Code

```
printf("Address of 'marks[4]' = %u", marks+4) ;
```

The above printf statement displays the address of element **"marks[4]"**.

Example Code

```
printf("Value of 'marks[0]' = %d", *marks) ;
printf("Value of 'marks[3]' = %d", *(marks+3)) ;
```

In the above two statements, first printf statement prints the value **89** (i.e., value of marks[0]) and the second printf statement prints the value **72** (i.e., value of marks[3]).

Example Code

```
marks++ ;
```

The above statement generates **compilation error** because the array name acts as a constant pointer. So we can't change its value.

In the above example program, the array name **marks** can be used as follows...

marks is same as **&marks[0]**

marks + 1 is same as **&marks[1]**

marks + 2 is same as **&marks[2]**

marks + 3 is same as **&marks[3]**

marks + 4 is same as **&marks[4]**

marks + 5 is same as **&marks[5]**

***marks** is same as **marks[0]**

***(marks + 1)** is same as **marks[1]**

***(marks + 2)** is same as **marks[2]**

***(marks + 3)** is same as **marks[3]**

***(marks + 4)** is same as **marks[4]**

***(marks + 5)** is same as **marks[5]**

Pointers to Multi Dimensional Array

In case of multi dimensional array also the array name acts as a constant pointer to the base address of that array. For example, we declare an array as follows...

Example Code

```
int  marks[3][3] ;
```

In the above example declaration, the array name **marks** acts as constant pointer to the base address (**address of marks[0][0]**) of that array.

In the above example of two dimensional array, the element **marks[1][2]** is accessed as ***(*(marks + 1) + 2)**.

Structure in C

Structure in c language is *a user defined datatype* that allows you to hold different type of elements.

Each element of a structure is called a member.

It works like a template in C++ and class in Java. You can have different type of elements in it.

It is widely used to store student information, employee information, product information, book information etc.

Defining structure

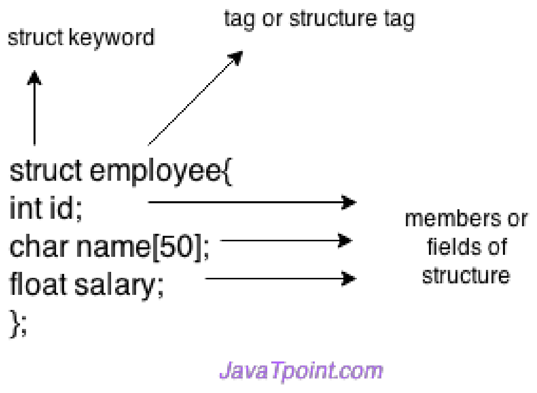
The **struct** keyword is used to define structure. Let's see the syntax to define structure in c.

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

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

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

Here, **struct** is the keyword, **employee** is the tag name of 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 variable for the structure, so that we can access the member of structure easily. There are two ways to declare structure variable:

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

1st way:

Let's see the example to declare 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;
```

2nd way:

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

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

Which approach is good

But if no. of variable are not fixed, use 1st approach. It provides you flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare variable in main() fuction.

Example:

```
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1, b2, b3 ;
```

is same as...

```
struct book
{
char name ;
float price ;
int pages ;
} b1, b2, b3 ;

or even...
struct{
char name ;
float price ;
int pages ;
} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
char name[10] ;
float price ;
int pages ;
} ;
struct book b1 = { "Basic", 130.00, 550 } ;

struct book b2 = { "Physics", 150.80, 800 } ;
```

How Structure Elements are Stored

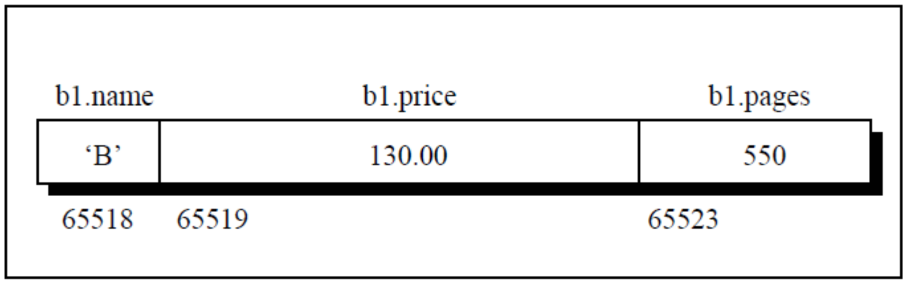
Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */
void main( )
{
struct book
{
char name ;
float price ;
int pages ;
} ;
struct book b1 = { 'B', 130.00, 550 } ;

printf ( "\nAddress of name = %u", &b1.name ) ;

printf ( "\nAddress of price = %u", &b1.price ) ;
printf ( "\nAddress of pages = %u", &b1.pages ) ;
}
```

Here is the output of the program...

```
Address of name = 65518
Address of price = 65519
Address of pages = 65523
```

Actually the structure elements are stored in memory as shown in the Figure 10.1.



Accessing members of structure

There are two ways to access structure members:

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
```

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);
   return 0;
}
```

Output:

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

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

```
#include <stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
{
   //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);

   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
```

Array of Structures in C

There can be array of structures in C programming to store many information of different data types. The array of structures is also known as collection of structures.

Let's see an example of structure with array that stores information of 5 students and prints it.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct student{
int rollno;
char name[10];
};
void main(){
int i;
struct student st[5];
clrscr();
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("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}

getch();
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

Nested Structure in C

Nested structure in c language can have another structure as a member. There are two ways to define nested structure in c language:

1. By separate structure
2. By Embedded structure

1) Separate structure

We can create 2 structures, but dependent structure should be used inside the main structure as a member. Let's see the code of nested structure.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

We can define structure within the structure also. It requires less code than previous way. But it can't be used in many structures.

```
struct Employee
{
  int id;
```

```
        char name[20];
        struct Date
         {
          int dd;
          int mm;
          int yyyy;
         }doj;
    }emp1;
```

Accessing Nested Structure

We can access the member of nested structure by Outer_Structure.Nested_Structure.member as given below:

```
    e1.doj.dd
    e1.doj.mm
    e1.doj.yyyy
```

C Nested Structure example

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

```
#include <stdio.h>
#include <string.h>
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
    int dd;
    int mm;
    int yyyy;
   }doj;
}e1;
int main( )
{
  //storing employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  e1.doj.dd=10;
  e1.doj.mm=11;
  e1.doj.yyyy=2014;
  //printing first employee information
  printf( "employee id : %d\n", e1.id);
  printf( "employee name : %s\n", e1.name);
  printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
  return 0;
}
```

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Like an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go. Let us examine both the approaches one by one using suitable programs.

```
/* Passing individual structure elements */
voidmain( )
{
struct book
{
char name[25] ;
char author[25] ;
intcallno ;
} ;
struct book b1 = { "Let us C", "YPK", 101 } ;
display ( b1.name, b1.author, b1.callno ) ;
}
display ( char *s, char *t, int n )
{
printf ( "\n%s %s %d", s, t, n ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Observe that in the declaration of the structure, **name** and **author** have been declared as arrays. Therefore, when we call the function **display( )** using,

```
display ( b1.name, b1.author, b1.callno ) ;
```

we are passing the base addresses of the arrays **name** and **author**, but the value stored in **callno**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time. This method is shown in the following program.

```
struct book
{
```

```
char name[25] ;
char author[25] ;
intcallno ;
} ;
voidmain( )
{
struct book b1 = { "Let us C", "YPK", 101 } ;
display ( b1 ) ;
}
display ( struct book b )
{
printf ( "\n%s %s %d", b.name, b.author, b.callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

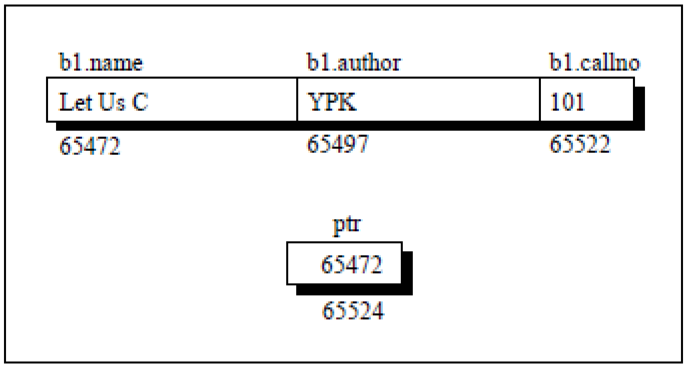
Note that here the calling of function **display( )** becomes quite compact,

```
display ( b1 ) ;
```

The way we can have a pointer pointing to an **int**, or a pointer pointing to a **char**, similarly we can have a pointer pointing to a **struct**. Such pointers are known as ‘structure pointers’. Let us look at a program that demonstrates the usage of a structure pointer.

```
voidmain( )
{
struct book
{
char name[25] ;
char author[25] ;
intcallno ;
} ;
struct book b1 = { "Let us C", "YPK", 101 } ;
struct book *ptr ;
ptr = &b1 ;
printf ( "\n%s %s %d", b1.name, b1.author, b1.callno ) ;
printf ( "\n%s %s %d", ptr->name, ptr->author, ptr->callno ) ;
}
```

The first **printf( )** is as usual. The second **printf( )** however is peculiar. We can’t use **ptr.name** or **ptr.callno**because **ptr**is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator **->**, called an arrow operator to refer to the structure elements. Remember that on the left hand side of the **‘.’** structure operator, there must always be a structure variable, whereas on the left hand side of the **‘->’** operator there must always be a pointer to a structure. The arrangement of the structure variable and pointer to structure in memory is shown in the Figure



Can we not pass the address of a structure variable to a function? We can. The following program demonstrates this.

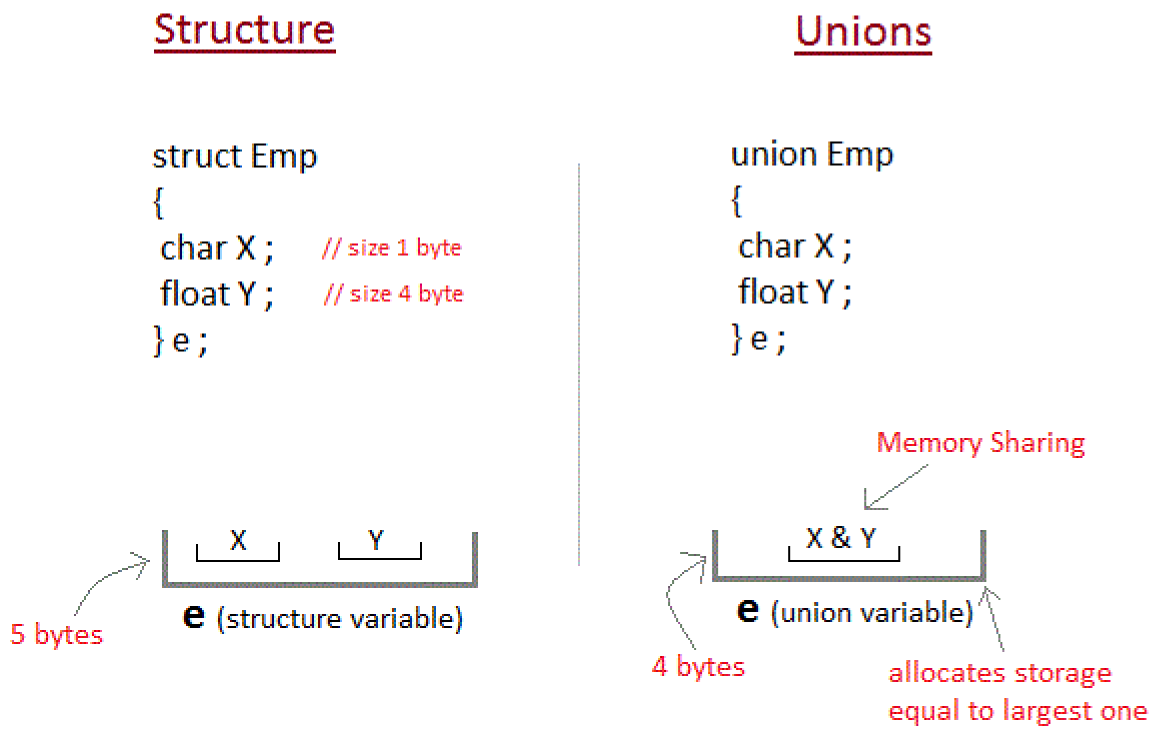
```
/* Passing address of a structure variable */
struct book
{
char name[25] ;
char author[25] ;
intcallno ;
} ;
void main( )
{
struct book b1 = { "Let us C", "YPK", 101 } ;
display ( &b1 ) ;
}
display ( struct book *b )
{
printf ( "\n%s %s %d", b->name, b->author, b->callno ) ;
}
```

And here is the output...

```
Let us C YPK 101
```

Unions

Unions are conceptually similar to **structures**. The syntax of **union** is also similar to that of structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



Advantage of union over structure

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

Disadvantage of union over structure

It can **store data in one member only**.

Defining union

The **union** keyword is used to define 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 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.

- Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	**Using pointer variable**

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

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

Declaring union using normal variable: union student report;	**Declaring union using pointer variable:** union student *report, rep;
Initializing union using normal variable:	**Initializing union using pointer variable:**

union student report = {100, “Mani”, 99.5};	union student rep = {100, “Mani”, 99.5};
	report = &rep;

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

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

union student
{
        char name[20];
        char subject[20];
        float percentage;
};

int main()
{
   union student record1;
   union student record2;

   // assigning values to record1 union variable
     strcpy(record1.name, "Raju");
     strcpy(record1.subject, "Maths");
     record1.percentage = 86.50;

     printf("Union record1 values example\n");
     printf(" Name       : %s \n", record1.name);
     printf(" Subject    : %s \n", record1.subject);
     printf(" Percentage : %f \n\n", record1.percentage);

   // assigning values to record2 union variable
     printf("Union record2 values example\n");
     strcpy(record2.name, "Mani");
     printf(" Name       : %s \n", record2.name);

     strcpy(record2.subject, "Physics");
     printf(" Subject    : %s \n", record2.subject);

     record2.percentage = 99.50;
     printf(" Percentage : %f \n", record2.percentage);
     return 0;
}
```

OUTPUT:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

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

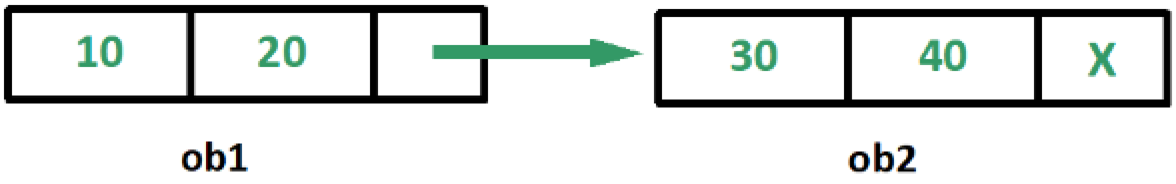
```
structnode {

    intdata1;

    chardata2;

    structnode* link;

};



intmain()

{

    structnode ob;

    return0;

}
```

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.
An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



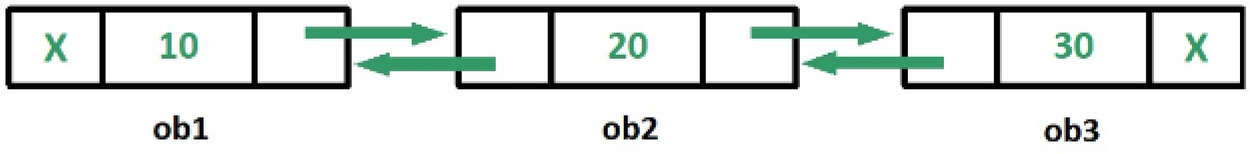
```
#include <stdio.h>

structnode {
    intdata1;
    chardata2;
    structnode* link;
};

intmain()
{
    structnode ob1; // Node1

    // Intialization
    ob1.link = NULL;
    ob1.data1 = 10;
    ob1.data2 = 20;

    structnode ob2; // Node2

    // Initialization
    ob2.link = NULL;
    ob2.data1 = 30;
    ob2.data2 = 40;

    // Linking ob1 and ob2
    ob1.link = &ob2;

    // Accessing data members of  ob2 using ob1
    printf("%d", ob1.link->data1);
    printf("\n%d", ob1.link->data2);
    return0;
}
```

Output:

```
30

40
```

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.

The connections made in the above example can be understood using the following figure.



```
#include <stdio.h>

structnode {
    intdata;
    structnode* prev_link;
    structnode* next_link;
};

intmain()
{
    structnode ob1; // Node1

    // Intialization
    ob1.prev_link = NULL;
    ob1.next_link = NULL;
    ob1.data = 10;

    structnode ob2; // Node2

    // Intialization
    ob2.prev_link = NULL;
    ob2.next_link = NULL;
    ob2.data = 20;

    structnode ob3; // Node3

    // Intialization
    ob3.prev_link = NULL;
    ob3.next_link = NULL;
    ob3.data = 30;

    // Forward links
    ob1.next_link = &ob2;
    ob2.next_link = &ob3;

    // Backward links
    ob2.prev_link = &ob1;
    ob3.prev_link = &ob2;

    // Accessing  data of ob1, ob2 and ob3 by ob1
    printf("%d\t", ob1.data);
    printf("%d\t", ob1.next_link->data);
    printf("%d\n", ob1.next_link->next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob2
    printf("%d\t", ob2.prev_link->data);
    printf("%d\t", ob2.data);
    printf("%d\n", ob2.next_link->data);

    // Accessing data of ob1, ob2 and ob3 by ob3
    printf("%d\t", ob3.prev_link->prev_link->data);
    printf("%d\t", ob3.prev_link->data);
    printf("%d", ob3.data);
    return0;
}
```
Run on IDE

Output:

```
10    20    30

10    20    30
```

```
10    20    30
```

In the above example we can see that 'ob1', 'ob2' and 'ob3' are three objects of the self referential structure 'node'. And they are connected using their links in such a way that any of them can easily access each other's data. This is the beauty of the self referential structures. The connections can be manipulated according to the requirements of the programmer.

Applications:
Self referential structures are very useful in creation of other complex data structures like:

- Linked Lists
- Stacks
- Queues
- Trees
- Graphs etc

C Programming Enumeration

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword `enum` is used.

```
enum flag { const1, const2, ..., constN };
```

Here, name of the enumeration is *flag*.

And, *const1*, *const2*,...., *constN* are values of type *flag*.

By default, *const1* is 0, *const2* is 1 and so on. You can change default values of enum elements during declaration (if necessary).

```
// Changing default values of enum

enum suit {

club = 0,

diamonds = 10,

hearts = 20,

spades = 3,
};
```

Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enumboolean { false, true };

enumboolean check;
```

Here, a variable *check* of type `enumboolean` is created.

Here is another way to declare same *check* variable using different syntax.

```
enumboolean

{

false, true

} check;
```

Example: Enumeration Type

```
#include<stdio.h>


enum week { sunday, monday, tuesday, wednesday, thursday, friday, saturday };


int main()
{
enum week today;
today = wednesday;
printf("Day %d",today+1);
return0;
}
```

Output

```
Day 4
```