**CHAPTER -8**

**Pointers**

**Introduction**

A pointer is a variable that stores the address of another variable. Memory can be visualized as an ordered sequence of consecutively numbered storage locations. A data item stored in memory in one or more adjacent storage locations depends upon its type. That is the number of memory locations required depends upon the type of a variable. The address of a data item is the address of its first storage location. This address can be stored in another data item and manipulated in a program. The address of a data item is a pointer to the data item, and a variable that holds the address is called a pointer variable. Some uses of pointers are:

1. Accessing array elements.
2. Returning more than one value from a function.
3. Accessing dynamically allocated memory.
4. Implementing data structures like linked lists, trees, and graphs.

### Address (&) Operator and indirection (\*) operator

C provides an address operator ‘&’, which returns the address of a variable when placed before it. This operator can be read as “the address of”, so &age means address of age, similarly &sal means address of sal. The following program prints the address of variables using address operator. We can access a variable indirectly using pointers. For this we will use the indirection operator (\*). By placing the indirection operator before a pointer variable, we can access the variable whose address is stored in the pointer.

&: Address of

\* : value at address of

Program to understand & and \* operator.

#include<stdio.h>

#include<conio.h>

void main()

{

int age=25,\*page;

float sal=50000,\*psal;

page=&age;

psal=&sal;

printf("\nAddress of age=%u and value of age=%d",page,\*page);

printf("\nAddress of sal=%u and value of sal=%f",psal,\*psal);

getch();

}

**Declaration of pointer**

Like all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location.

Syntax:

datatype \*identifier

eg:

int age; //declaring a normal variable

int \*ptr; //declaring a pointer variable

ptr=&age; //store the address of the variable age in the variable ptr.

* Now ptr points to age or ptr is a pointer to age.
* We’ve created a pointer ptr which becomes a variable that contains the address of another variable age.

65524

25

ptr

65528

65524

age

**Pointer arithmetic**

All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as:

1. Addition of an integer to a pointer and increment operation.
2. Subtraction of an integer from a pointer and decrement operation.
3. Subtraction of a pointer from another pointer of same type.

Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of pointer. For example if we have an integer pointer pi which contains address 1000 then on incrementing we get 1002 instead of 1001. This is because the size of int data type is 2. Similarly on decrementing pi, we will get 998 instead of 999. The expression (pi + 3) will represent the address 1006.

**Program to understand pointer arithmetic**

#include<stdio.h>

#include<conio.h>

void main()

{

int a=5,\*pi;

float b=7.3,\*pf;

char c='x',\*pc;

clrscr();

pi=&a;

pf=&b;

pc=&c;

printf("\n Address of a=%u ",pi);

printf("\n Address of b=%u ",pf);

printf("\n Address of c=%u ",pc);

pi++;

pf++;

pc++;

printf("\n\n Address of a=%u ",pi);

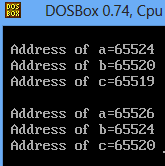
printf("\n Address of b=%u ",pf);

printf("\n Address of c=%u ",pc);

getch();

}

Output



The arithmetic operations that can never be performed on pointers are:

1. Addition, multiplication, division of two pointers.
2. Multiplication between pointer and any number.
3. Division of a pointer by any number.
4. Addition of float or double values to pointers.

**pointer to pointer (double indirection)**

Pointer variable itself might be another pointer so pointer which contains another pointer’s address is called pointers to pointers or multiple indirections.

Syntax

datatype \*\*identifier;

Eg:

int \*\*pptr;

Program to understand pointer to pointer

#include<stdio.h>

#include<conio.h>

void main()

{

int a=5,\*pa,\*\*ppa;

clrscr();

pa=&a;

ppa=&pa;

printf("\nValue of a=%d",\*pa);

printf("\nValue of a=%d",\*\*ppa);

getch();

}

1028

ppa

1030

1024

pa

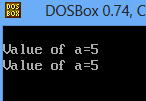
1028

5

a

1024

Output



**Array and Pointer**

In C, there is a strong relationship between arrays and pointers. Any operations that can be achieved by array subscripting can also be done with pointers. The pointer version will, in general, be faster but for the beginners somewhat harder to understand.

An array name is itself is an address, or pointer value, so any operation that can be achieved by array subscripting can also be done with pointers as well. Pointers and arrays are almost synonymous in terms of how they are used to access memory, but there are some important differences between them. A pointer which is fixed. An array name is a constant pointer to the first element of the array. The relation of pointer and array is presented in the following tables:

|  |  |
| --- | --- |
| **Address of array elements** | |
| Address of array elements | Equivalent pointer notation |
| &arr[0] | arr |
| &arr[1] | (arr+1) |
| &arr[i] | (arr+i) |

|  |  |
| --- | --- |
| **Value of array elements** | |
| Value of array elements | Equivalent pointer notation |
| arr[0] | \*arr |
| arr[1] | \*(arr+1) |
| arr[i] | \*(arr+i) |
| arr[i][j] | \*(\*(arr+i)+j) |

**Pointer and one dimensional array**

The elements of an array are stored in contiguous memory locations. Suppose we have an array arr[5] of type int.

int arr[5]={6,9,12,4,8};

This is stored in memory as-

arr[0]

arr[1]

arr[2]

arr[3]

arr[4]

6

9

12

4

8

5000

5002

5004

5006

5008

Here 5000 is the address of first element, and since each element (type int) takes 2 bytes so address of next element is 5002, and so on. The address of first element of the array is also known as the base address of the array.

Program to print the value of array element using pointer.

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[100],i,n;

clrscr();

printf("How many elements are there: ");

scanf("%d",&n);

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

{

printf("\n Enter element: ");

scanf("%d",(arr+i));

}

printf("\n Printing array element:");

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

{

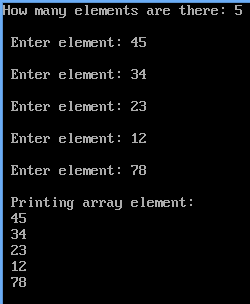
printf("\n %d",\*(arr+i));

}

getch();

}

**Output**



Examples

1. Write a program to sort ‘n’ numbers in ascending order using pointer.

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[100],n,i,j,temp;

clrscr();

printf("How many elements are there?: ");

scanf("%d",&n);

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

{

printf("\n Enter elements :");

scanf("%d",(arr+i));

}

for(i=0;i<n-1;i++)

{

for(j=i+1;j<n;j++)

{

if(\*(arr+i)>\*(arr+j))

{

temp=\*(arr+i);

\*(arr+i)=\*(arr+j);

\*(arr+j)=temp;

}

}

}

printf("\n Printing sorted elements:");

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

{

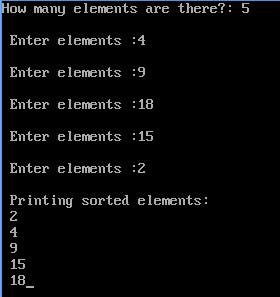
printf("\n %d",\*(arr+i));

}

getch();

}

**output**



1. Write a program to find sum of all the elements of an array using pointers.

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[100],n,i,sum=0;

clrscr();

printf("How many elements are there? ");

scanf("%d",&n);

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

{

printf("\n Enter elements of array: ");

scanf("%d",(arr+i));

}

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

{

sum=sum + \*(arr+i);

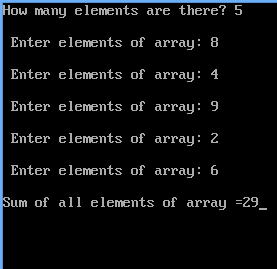
}

printf("\nSum of all elements of array =%d",sum);

getch();

}

**output**



1. Write a program to search an element from ‘n’ number of elements using pointer.

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[100],n,i,search;

clrscr();

printf("How many elements are there? ");

scanf("%d",&n);

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

{

printf("\n Enter elements :");

scanf("%d",(arr+i));

}

printf("\n Enter an element to be searched: ");

scanf("%d",&search);

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

{

if(search==\*(arr+i))

{

printf("\n %d is found in %d position",search,i+1);

break;

}

}

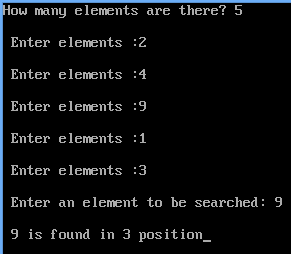
if(i==n)

printf("\n %d is not found",search);

getch();

}

**Output**



**Pointer with two dimensional arrays**

In a two dimensional array we can access each element by using two subscripts, where first subscript represents row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation. Suppose arr is a 2-D array, then we can access any element arr[i][j] of this array using the pointer expression \*(\*(arr+i)+j).

Example

1. Write a program to read any 2 by 3 matrix and display its element in appropriate format.

#include<stdio.h>

#include<conio.h>

void main()

{

int arr[2][3],i,j;

clrscr();

for(i=0;i<2;i++)

{

for(j=0;j<3;j++)

{

printf("\n Enter elements of matrix: ");

scanf("%d",&arr[i][j]);

}

}

for(i=0;i<2;i++)

{

for(j=0;j<3;j++)

{

printf("%d\t",\*(\*(arr+i)+j));

}

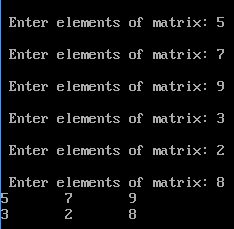
printf("\n");

}

getch();

}

**Output**



**Pointer and strings**

We can take a char pointer and initialize it with a string constant. For example:

char \*ptr=”Programming”;

Here ptr is a char pointer which points to the first character of the string constant “Programming” ie. ptr contains the base address of this string constant.

Now let’s compare the strings defined as arrays and strings defined as pointers.

char str[]=”Chitwan”;

char \*ptr=”Nawalparasi”;

These two forms may look similar but there are some differences in them. The initialization itself has different meaning in both forms. In array form, initialization is a short form for-

char str[]={‘C’,’h’,’i’,’t’,’w’,’a’,’n’,’\0’}

while in pointer form, address of string constant is assigned to the pointer variable.

Now let us see how they are represented in memory.

C

h

i

t

w

a

n

\0

1000

1001

1002

1003

1004

1005

1006

1007

str[0]]

str[1]]

str[2]]

str[3]]

str[4]]

str[5]]

str[6]]

str[7]]

5220 5221

200

ptr

N

a

w

a

l

p

a

r

a

s

i

\0

200

201

202

203

204

205

206

207

208

209

210

211

Here string assignments are valid for pointers while they are invalid for strings defined as arrays.

str = “Narayangarh” //invalid

ptr = ”Butwal” //valid

**Array of pointers with string**

Array of pointers to strings is an array of char pointers in which each pointer points to the first character of a string i.e. each element of this array contains the base address of a string.

char \*arrp[]={“white”,”red”,”green”,”yellow”,”blue”};

Here arrp is an array of pointers to string. We have not specified the size of array, so the size is determined by the number of initializers. The initializers are string constant. arrp[0] contains the base address of string “white” similarly arrp[1] contains the base address of string “red”.

w

h

i

t

e

\0

r

e

d

\0

g

r

e

e

n

\0

y

e

l

l

o

b

l

u

e

\0

106

100

110

116

123

w

\0

100

106

110

116

123

arrp[0]

arrp[1]

arrp[2]

arrp[3]

arrp[4]

2000

String: white Address of string : 100 Address of string is stored at : 2000

String: red Address of string : 106 Address of string is stored at : 2002

String: green Address of string : 110 Address of string is stored at : 2004

String: yellow Address of string : 116 Address of string is stored at : 2006

String: blue Address of string : 123 Address of string is stored at : 2008

**Pointers as Function Arguments**

Just like any other argument, pointers can also be passed to a function as an argument. Lets take an example to understand how this is done.

Example: Passing Pointer to a Function in C Programming

In this example, we are passing a pointer to a function. When we pass a pointer as an argument instead of a variable then the address of the variable is passed instead of the value. So any change made by the function using the pointer is permanently made at the address of passed variable. This technique is known as call by reference in C.

Try this same program without pointer, you would find that the bonus amount will not reflect in the salary, this is because the change made by the function would be done to the local variables of the function. When we use pointers, the value is changed at the address of variable

#include <stdio.h>

void salaryhike(int \*var, int b)

{

\*var = \*var+b;

}

int main()

{

int salary=0, bonus=0;

printf("Enter the employee current salary:");

scanf("%d", &salary);

printf("Enter bonus:");

scanf("%d", &bonus);

salaryhike(&salary, bonus);

printf("Final salary: %d", salary);

return 0;

}

**Output**

Enter the employee current salary:10000

Enter bonus:2000

Final salary: 12000

Example 2: Swapping two numbers using Pointers

This is one of the most popular example that shows how to swap numbers using call by reference.

Try this program without pointers, you would see that the numbers are not swapped. The reason is same that we have seen above in the first example.

#include <stdio.h>

void swapnum(int \*num1, int \*num2)

{

int tempnum;

tempnum = \*num1;

\*num1 = \*num2;

\*num2 = tempnum;

}

int main( )

{

int v1 = 11, v2 = 77 ;

printf("Before swapping:");

printf("\nValue of v1 is: %d", v1);

printf("\nValue of v2 is: %d", v2);

/\*calling swap function\*/

swapnum( &v1, &v2 );

printf("\nAfter swapping:");

printf("\nValue of v1 is: %d", v1);

printf("\nValue of v2 is: %d", v2);

}

**Output**

Before swapping:

Value of v1 is: 11

Value of v2 is: 77

After swapping:

Value of v1 is: 77

Value of v2 is: 11

**Function returning pointers**

C also allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example −

int \* myFunction()

{

.

.

.

}

Second point to remember is that, it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as **static** variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

[Live Demo](http://tpcg.io/cWvLL4)

#include <stdio.h>

#include <time.h>

/\* function to generate and return random numbers. \*/

int \* getRandom( ) {

static int r[10];

int i;

/\* set the seed \*/

srand( (unsigned)time( NULL ) );

for ( i = 0; i < 10; ++i) {

r[i] = rand();

printf("%d\n", r[i] );

}

return r;

}

/\* main function to call above defined function \*/

int main () {

/\* a pointer to an int \*/

int \*p;

int i;

p = getRandom();

for ( i = 0; i < 10; i++ ) {

printf("\*(p + [%d]) : %d\n", i, \*(p + i) );

}

return 0;

}

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

1523198053

1187214107

1108300978

430494959

1421301276

930971084

123250484

106932140

1604461820

149169022

\*(p + [0]) : 1523198053

\*(p + [1]) : 1187214107

\*(p + [2]) : 1108300978

\*(p + [3]) : 430494959

\*(p + [4]) : 1421301276

\*(p + [5]) : 930971084

\*(p + [6]) : 123250484

\*(p + [7]) : 106932140

\*(p + [8]) : 1604461820

\*(p + [9]) : 149169022

# Pointer to a Structure

We have already learned that a pointer is a variable which points to the address of another variable of any data type like int, char, float etc. Similarly, we can have a pointer to structures, where a pointer variable can point to the address of a structure variable. Here is how we can declare a pointer to a structure variable.

struct dog

{

char name[10];

char breed[10];

int age;

char color[10];

};

struct dog spike;

// declaring a pointer to a structure of type struct dog

struct dog \*ptr\_dog

This declares a pointer ptr\_dog that can store the address of the variable of type struct dog. We can now assign the address of variable spike to ptr\_dog using & operator.

ptr\_dog = &spike;

Now ptr\_dog points to the structure variable spike.

**Accessing members using Pointer**

There are two ways of accessing members of structure using pointer:

Using indirection (\*) operator and dot(.) operator.  
1 .Using arrow (->) operator or membership operator.

Let's start with the first one.

Using Indirection (\*) Operator and Dot(.) Operator

At this point ptr\_dog points to the structure variable spike, so by dereferencing it we will get the contents of the spike. This means spike and \*ptr\_dog are functionally equivalent. To access a member of structure write \*ptr\_dog followed by a dot(.) operator, followed by the name of the member. For example:

(\*ptr\_dog).name - refers to the name of dog  
(\*ptr\_dog).breed - refers to the breed of dog

and so on.

Parentheses around \*ptr\_dog are necessary because the precedence of dot(.) operator is greater than that of indirection (\*) operator.

Using arrow operator (->)

The above method of accessing members of the structure using pointers is slightly confusing and less readable, that's why C provides another way to access members using the arrow (->) operator. To access members using arrow (->) operator write pointer variable followed by -> operator, followed by name of the member.

ptr\_dog->name - refers to the name of dog

ptr\_dog->breed - refers to the breed of dog

and so on.

Here we don't need parentheses, asterisk(\*) and dot(.) operator. This method is much more readable and intuitive.

We can also modify the value of members using pointer notation.

strcpy(ptr\_dog->name, "new\_name");

Here we know that the name of the array (ptr\_dog->name) is a constant pointer and points to the 0th element of the array. So we can't assign a new string to it using assignment operator(=), that's why strcpy() function is used.

--ptr\_dog->age;

In the above expression precedence of arrow operator (->) is greater than that of prefix decrement operator (--), so first -> operator is applied in the expression then its value is decremented by 1.

The following program demonstrates how we can use a pointer to structure.

#include<stdio.h>

struct dog

{

char name[10];

char breed[10];

int age;

char color[10];

};

int main()

{

struct dog my\_dog = {"tyke", "Bulldog", 5, "white"};

struct dog \*ptr\_dog;

ptr\_dog = &my\_dog;

printf("Dog's name: %s\n", ptr\_dog->name);

printf("Dog's breed: %s\n", ptr\_dog->breed);

printf("Dog's age: %d\n", ptr\_dog->age);

printf("Dog's color: %s\n", ptr\_dog->color);

// changing the name of dog from tyke to jack

strcpy(ptr\_dog->name, "jack");

// increasing age of dog by 1 year

ptr\_dog->age++;

printf("Dog's new name is: %s\n", ptr\_dog->name);

printf("Dog's age is: %d\n", ptr\_dog->age);

// signal to operating system program ran fine

return 0;

}

**Expected Output:**

Dog's name: tyke

Dog's breed: Bulldog

Dog's age: 5

Dog's color: white

After changes

Dog's new name is: jack

Dog's age is: 6

**Dynamic memory allocation**

Dynamic memory allocation is necessary to manage available memory. For example, during compile time, we may not know the exact memory needs to run the program. So for the most part, memory allocation decisions are made during the run time. C also does not have automatic garbage collection like Java does. Therefore a C programmer must manage all dynamic memory used during the program execution. The <stdlib.h> provides four functions that can be used to manage dynamic memory. calloc, malloc, free, realloc - Allocate and free dynamic memory.

The memory allocation that we have done till now was static memory allocation. The memory that could be used by the program was fixed i.e. we could not increase or decrease the size of memory during the execution of program. In many applications it is not possible to predict how much memory would be needed by the program at run time. For example if we declare an array of integers.

int emp[100];

In an array, it is must to specify the size of array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur.

1. The number of values to be stored is less than the size of array then there will be the wastage of memory.
2. If we want to store more values than the size of array then we can’t.

To overcome these problems we should be able to allocate memory at run time. The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in alloc.h and stdlib.h header files.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

1. malloc ( )

This function is used to allocate memory dynamically. The malloc function allocates a memory block of size n bytes (size\_t is equivalent to an unsigned integer) The malloc function returns a pointer (void\*) to the block of memory. That void\* pointer can be used for any pointer type. malloc allocates a contiguous block of memory. If enough contiguous memory is not available, then malloc returns NULL.

syntax:

pointer\_variable=(datatype\*) malloc(specified\_size);

Here pointer\_variable is a pointer of type datatype, and specified\_size is the size in bytes required to be reserved in memory.

Example: if we need an array of n ints, then we can do

int\* A = malloc(n\*sizeof(int));

A holds the address of the first element of this block of 4n bytes, and A can be used as an array.

For example,

if (A != NULL)

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

A[i] = 0;

will initialize all elements in the array to 0. We note that A[i] is the content at address (A+i). Therefore we can also write

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

\*(A+i) = 0;

Recall that A points to the first byte in the block and A+i points to the address of the ith element in the list. That is &A[i]. We can also see the operator [] is equivalent to doing pointer arithmetic to obtain the content of the address.

A dynamically allocated memory can be freed using free function. For example

free(A);

will cause the program to give back the block to the heap (or free memory). The argument to free is any address that was returned by a prior call to malloc. If free is applied to a location that has been freed before, a double free memory error may occur. We note that malloc returns a block of void\* and therefore can be assigned to any type.

double\* A = (double\*)malloc(n);

int\* B = (int\*)malloc(n);

char\* C = (char\*)malloc(n);

In each case however, the addresses A+i, B+i, C+i are calculated differently.

• A + i is calculated by adding 8i bytes to the address of A (assuming sizeof(double) = 8)

• B + i is calculated by adding 4i bytes to the address of B

• C + i is calculated by adding i bytes to the address of C

1. calloc ( )

The calloc ( ) function is used to allocate multiple blocks of memory. It is somewhat similar to malloc ( ) function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example:

ptr= (int \*) calloc (5, sizeof(int));

The other difference between calloc( ) and malloc( ) is that the memory allocated by malloc( ) contains garbage value while the memory allocated by calloc( ) is initialized to zero.

1. realloc( )

The function realloc( ) is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc( ) or calloc( ) and second one is the new size for that block. For example

ptr=(int\*) realloc(ptr,newsize);

1. Program to understand dynamic allocation of memory.

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

void main()

{

int \*ptr,n,i;

clrscr();

printf("How many numbers do you want to entered: ");

scanf("%d",&n);

ptr=(int \*)malloc(n\*sizeof(int));

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

{

printf("\n Enter number: ");

scanf("%d",ptr+i);

}

printf("\nDisplaying elements: ");

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

{

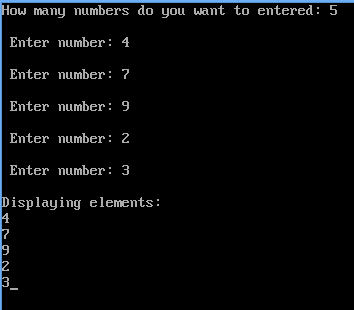
printf("\n%d",\*(ptr+i));

}

getch();

}

**Output**

****

1. Write a program to sort ‘n’ numbers in ascending order using dynamic memory.

#include<stdio.h>

#include<conio.h>

#include<alloc.h>

void main()

{

int \*ptr,n,i,j,temp;

clrscr();

printf("How many numbers do you want to entered: ");

scanf("%d",&n);

ptr=(int \*)malloc(n\*sizeof(int));

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

{

printf("\n Enter number: ");

scanf("%d",ptr+i);

}

for(i=0;i<n-1;i++)

{

for(j=i+1;j<n;j++)

{

if(\*(ptr+i)>\*(ptr+j))

{

temp=\*(ptr+i);

\*(ptr+i)=\*(ptr+j);

\*(ptr+j)=temp;

}

}

}

printf("\nDisplaying sorted elements: ");

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

{

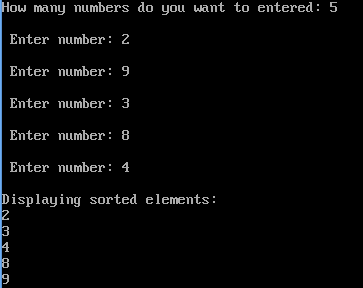
printf("\n%d",\*(ptr+i));

}

getch();

}

**Output**

****