

Chapter 9

Pointers

Pointer is a powerful technique to access the data by indirect reference as it holds the address of that variable where it has been stored in memory. A pointer is a variable, which holds the memory address of another variable. Sometimes, only with the pointer a complex data type can be declared and accessed easily. The pointer has the following advantages.

- It enhances the execution speed of a program.
- It makes the programs simple and reduce their length.
- It is helpful in traversing through arrays and character strings. The strings are also arrays of characters terminated by the null character ('\0').
- Pointer also acts as references to different types of objects such as variables, arrays, functions, structures, etc. However, C language does not have the concept of references as in C++. Therefore, in C we use pointer as a reference.
- Storage of strings through pointers saves memory space.
- It allows passing variables, arrays, strings and structures as function arguments.
- It is used to construct different data structures such as linked lists, queues, stacks, etc.
- A pointer allows returning structures, variables from functions.
- It provides functions which can modify their calling arguments.
- Pointer can return more than one value.
- It supports dynamic allocations and de-allocations of memory segments.
- Passing on arrays by pointers saves lot of memory because we are passing on only the address of array instead of all the elements of an array, which would mean passing on copies of all the elements and thus taking lot of memory space.

A pointer contains the memory address. Most commonly this address is the locations of another variable, where it has been stored in memory. If one variable contains the address of another variable then the first variable is said to point to the second. In C and C++ pointer are distinct such as integer pointer, character pointer, floating point number pointer etc. A pointer consists of two parts, namely: (a) Pointer Operator and (b) Address Operator

a) Pointer Operator

A **pointer operator** can be represented by combinations of * (*asterisk*) with a variable. For e.g. if a variable of integer data type and also declared * (asterisk) with another variable, it means the variable is of type “pointer to integer”. In other words, it will be used in the program indirectly to access the value of one or more integer variables. For Example:

```
int *myptr;
```

where myptr is a pointer, which holds the address of an integer data type. All pointer variables must be declared before it is used in C and C++ programs like other variables. When a pointer variables is declared, and asterisk must precede the variable name this identifies the *variable as a pointer*.

The general form of pointer declaration

```
data_type *pointer_variable_name;
```

b) Address Operator

An **address operator** can be represented by a combination of & (ampersand) with a pointer variable. For e.g. if a pointer variable is an integer type and also declared & with the pointer variable then it means that the variable is of type “address of”. In other words, it will be used in the programs to indirectly access the value of one or more integer variables. The “&” is unary operator that returns the memory address of its operand. A unary operator requires only one operand.

```
m=&ptr;
```

Note, that the pointer operator & is an operator that returns the address of the variable following it. Therefore, the preceding assignment statement could be verbalized as “m receives the address of ptr”.

The other operator * is the complement of &. It is unary operator that returns the value of the variable located at the address that follows. The operation of * implies to the phrase “at address”. Unfortunately, the symbol * represents both the multiplication sign and the “at address”, and the symbol & represents both bitwise AND and the “address of” sign. When these are used as pointer operators, they have no relationship to the arithmetic operators. That happened to look like the same. Both, the pointer operators, & and *, have higher precedence over all other arithmetic operators except the unary minus, with which they have equal precedence.

Pointer Assignment

A pointer is a variable data type and hence the general rule to assign its value to the pointer is same as that of any other variable data type. For e.g.

```
int x=5,y;
```

```
int *ptr1,*ptr2;
```

- 1) ptr1=&x; Here, the memory address of variables x is assigned to the pointer variable ptr1.
- 2) y=*ptr1; The contents of the pointer variables is assigned to the variable y, not the memory address
- 3) ptr2=ptr1; /*address if ptr1 is assigned to the ptr2*/ The address of the ptr2 is assigned to the pointer variable ptr2. The contents of both ptr1 and ptr2 will be same as these two pointer variables hold the same address

Graphical figure of the above demo

1)

x	ptr1
5	1000H
1000H	5000H

2)

y	ptr1
5	1000H
3000H	5002H

Here, the value or content of ptr1 i.e 100H (address) is assigned to another pointer. So, this both pointers show to the same variables.

3)

ptr2	ptr1
1000H	1000H
5002H	5000H

Example#1: A program to display the contents of the pointer

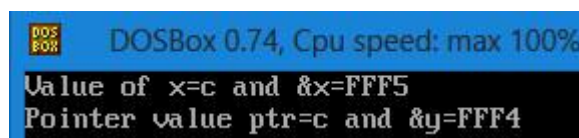
```
#include<stdio.h>
#include<conio.h>
void main(){
    int x;
    int *ptr;
    x=10;
    clrscr();
    ptr=&x;
    printf("x=%d and ptr=%X",x,ptr);
    printf("\nx=%d and contents of ptr=%d",x,*ptr);
    getch();
}
```

x
10
FFF4

**Example#2: Assign a character variable to the pointer and display the contents of the pointer**

```
#include<stdio.h>
#include<conio.h>
void main(){
    char x,y;
    char *ptr;
    x='c'; /*assignment of character*/
    clrscr();
    ptr=&x;
    y=*ptr;
    printf("Value of x=%c and &x=%X", x,ptr);
    printf("\nPointer value ptr=%c and &y=%X",y,&y);
    getch();
}
```

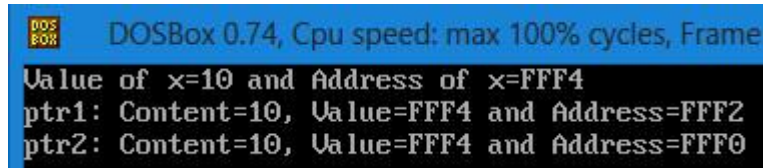
x	y
c	c
FFF5	FFF4

**Example#3: A program to assign the pointer variable to another pointer and display the contents of the both the pointer variables**

```
#include<stdio.h>
#include<conio.h>
void main(){
    int x;
    int *ptr1,*ptr2;
    clrscr();
    x=10;
    ptr1=&x;
    ptr2=ptr1;
    printf("Value of x=%d and Address of x=%X",x,&x);
    printf("\nptr1: Content=%d, Value=%X and Address=%X",*ptr1,ptr1,&ptr1);
}
```

```
printf("\nptr2: Content=%d, Value=%X and Address=%X", *ptr2, ptr2, &ptr2);
getch();
}
```

x	ptr1	ptr2
10	FFF4	FFF4
FFF4	FFF2	FFF0



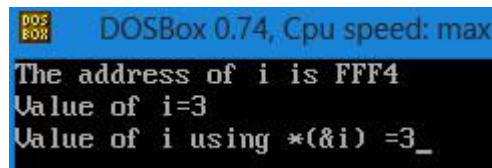
DOSBox 0.74, Cpu speed: max 100% cycles, Frame

Value of x=10 and Address of x=FFF4
 ptr1: Content=10, Value=FFF4 and Address=FFF2
 ptr2: Content=10, Value=FFF4 and Address=FFF0

Example#4: A program for assignment and usage of & operator

```
main(){
    int i=3;
    printf("The address of i is %X",&i);
    printf("\nValue of i=%d",i);
    printf("\nValue of i using *(&i) %d=",*(&i)); getch();
}
```

i
3
FFF4

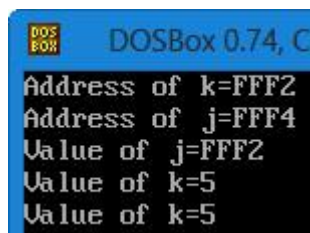


DOSBox 0.74, Cpu speed: max

The address of i is FFF4
 Value of i=3
 Value of i using *(&i) =3_

```
main(){
    int *j,k=5;
    j=&k;
    printf("Address of k=%X",&k);
    printf("\nAddress of j=%X",&j);
    printf("\nValue of j=%X",j);
    printf("\nValue of k=%d",k);
    printf("\nValue of k=%d",*j);
}
```

j	k
FFF2	5
FFF4	FFF2



DOSBox 0.74, C

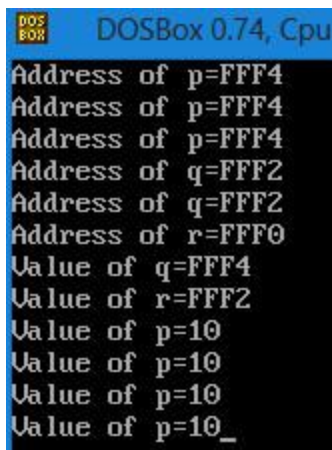
Address of k=FFF2
 Address of j=FFF4
 Value of j=FFF2
 Value of k=5
 Value of k=5

```

#include<stdio.h>
#include<conio.h>
void main(){
    int p=10,*q,**r;
    clrscr();
    q=&p;
    r=&q;
    printf("Address of p=%X",&p);
    printf("\nAddress of p=%X",q);
    printf("\nAddress of p=%X",*r);
    printf("\nAddress of q=%X",&q);
    printf("\nAddress of q=%X",r);
    printf("\nAddress of r=%X",&r);
    printf("\nValue of q=%X",q);
    printf("\nValue of r=%X",r);
    printf("\nValue of p=%d",p);
    printf("\nValue of p=%d",&p);
    printf("\nValue of p=%d",*q);
    printf("\nValue of p=%d",**r);
    getch();
}

```

p	q	r
10	FFF4	<i>FFF2</i>
FFF4	<i>FFF2</i>	FFF0



Pointer Arithmetic

As a pointer holds the memory address of variable, some arithmetic operations can be performed with pointers. C and C++ supports four arithmetic operators that can be used with pointers, such as

Addition	+
Subtraction	-
Increment	++
Decrement	--

Pointers are variables. They are not integers, but they can be displays as unsigned integers. The conversion specifier for a pointer is added and subtracted. For e.g.

`ptr++` causes the pointer to be incremented, but not by 1

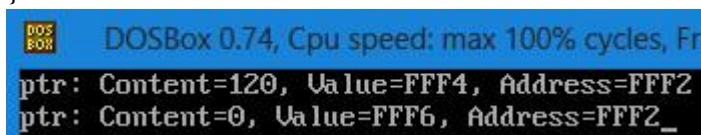
`ptr--` causes the pointer to be decremented, but not by 1

According to datatype declared to that pointer variable if arithmetic operation is done then values(contents) will be incremented or decremented as per the data type is chosen

- The following program segment illustrates the pointer arithmetic.
- The integer would occupy addresss FFF4 to FFF6 (two bytes of memory allocation)

Example#5: A program to increment the pointer's address

```
#include<stdio.h>
#include<conio.h>
void main(){
    int value,*ptr;
    clrscr();
    value=120;
    clrscr();
    ptr=&value;
    printf("ptr: Content=%d, Value=%X, Address=%X",*ptr,ptr,&ptr);
    ptr++;
    printf("\nptr: Content=%X, Value=%X, Address=%X",*ptr,ptr,&ptr);
    getch();
}
```



The screenshot shows the output of the program in a DOSBox 0.74 window. The first line of output is "ptr: Content=120, Value=FFF4, Address=FFF2". The second line of output is "ptr: Content=0, Value=FFF6, Address=FFF2_".

Here, lets say address of ptr which it is pointing is FFF4 then after `ptr++` is now increment not by one but by 2 as it is defined as integer and compiler knows that it should occupy 2 bytes show as first ti will show FFF4 and after `ptr++` it wil FFF6.

Example#6: A program to show the arithmetic operation on pointers

```
#include<stdio.h>
#include<conio.h>
void main(){
    int value,*p;
    int x,*p1,*p2;
    clrscr();
    printf("&value=%X, &p=%X, &x=%X, &p1=%X,
    &p2=%X",&value,&p,&x,&p1,&p2);
    printf("\n&value=%u, &p=%u, &x=%u, &p1=%u,
    &p2=%u\n",&value,&p,&x,&p1,&p2);
    value=120;
    p=&value;
    printf("\nValue of p=%X",p);
    p++;
    printf("\nMemory address after increment=%X",p);
    p--;
    printf("\nMemory address after decrement=%X",p);
    x=101;
    printf("\nx: Value=%d, Address=%X",x,&x);
    p1=&x;
    printf("\np1: Content=%d, Value=%X,
    Address=%X",*p1,p1,&p1);
    p2=p1+2; /* 2*2 Bytes */
    printf("\nAddress of (p2=p1+2)=%X",&p2);
    printf("\np2: Content=%X, Value=%X,
    Address=%X",*p2,p2,&p2);
    getch();
}
```

value	ptr	x	ptr1	ptr2
120	FFF4	101	FFF0	FFFC
FFF4	FFF2	FFF0	FFEE	FFEC
65524	65522	65520	65518	65516

Address	Memory Cell	Hex Address
0		0000
1		0001
⋮	⋮	⋮
65514		
65515	FFFC	FFEC
65516		
65517	FFF0	FFEE
65518		
65519	101	FFF0
65520		
65521	FFF4	FFF2
65522		
65523	120	FFF4
65524		
⋮	⋮	⋮
65534		FFFE
65535		FFFF

```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Pro
&value=FFF4, &p=FFF2, &x=FFF0, &p1=FFEE, &p2=FFEC
&value=65524, &p=65522, &x=65520, &p1=65518, &p2=65516

Value of p=FFF4
Memory address after increment=FFF6
Memory address after decrement=FFF4
x: Value=101, Address=FFF0
p1: Content=101, Value=FFF0, Address=FFEE
Address of (p2=p1+2)=FFEC
p2: Content=78, Value=FFF4, Address=FFEC_
```

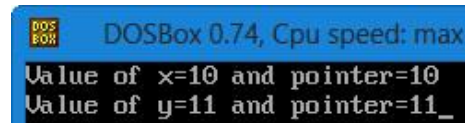
Example#7: Other examples using pointer operation

```

main(){
    int x,y,*ptr;
    x=10;
    ptr=&x;
    printf("Value of x=%d and pointer=%d",x,*ptr);
    y=++ *ptr; /*y=11*/
    printf("\nValue of y=%d and pointer=%d",y,*ptr);
    getch();
}

```

x	y	ptr
10	11	&x
&x	&y	&ptr



```

DOS
DOSBox 0.74, Cpu speed: max
Value of x=10 and pointer=10
Value of y=11 and pointer=11_

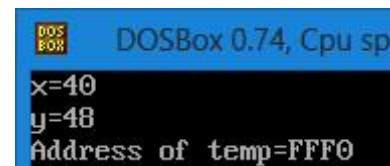
```

```

main(){
    int x,y,*x_pointer,temp;
    temp=3;
    x=5*(temp+5); /* 5*(3+5) */
    x_pointer=&temp;
    y=6*(x_pointer+5); /* 6*(3+5) */
    printf("x=%d",x);
    printf("\ny=%d",y);
    printf("\nAddress of temp=%X",&temp);
    getch();
}

```

x	y	x_pointer	temp
40	48	FFF0	3
			FFF0



```

DOS
DOSBox 0.74, Cpu sp
x=40
y=48
Address of temp=FFF0

```


Example#7.3:

```

#include<stdio.h>
#include<conio.h>
void main(){
    int x,y,*ptr,*ptr2;
    clrscr();
    x=25;
    ptr=&x;
    printf("&x=%X, &y=%X, &ptr=%X, &ptr2=%X\n",&x,&y,&ptr,&ptr2);
    printf("Contents of pointer=%d",*ptr);
    *ptr=*ptr+1; /* ++*ptr */
    y=*ptr;
    printf("\nx=%d, y=%d and (*ptr=*ptr+1)=%d",x,y,*ptr);
    (*ptr)++;
    y=*ptr;
    printf("\nx=%d, y=%d and (*ptr)++=%d",x,y,*ptr);
    ++ *ptr;
    y=*ptr;
    printf("\nx=%d, y=%d and ++ *ptr=%d",x,y,++ *ptr);
    ++ *ptr;
    ptr2=ptr;
    printf("\n\n*ptr=%d",*ptr);
    printf("\n*ptr2=%d",*ptr2);
    getch();
}

```

x	y	ptr	ptr2
25	26	&x	&x
&x	&y	&ptr	&ptr2

26 28
 27
 28
 29
 30

```

DOS
DOSBox 0.74, Cpu speed: max 100% cycles,
&x=FFF4, &y=FFF2, &ptr=FFF0, &ptr2=FFEE
Contents of pointer=25
x=26, y=26 and (*ptr=*ptr+1)=26
x=27, y=27 and (*ptr)++=27
x=29, y=28 and ++ *ptr=29

*ptr=30
*ptr2=30_
    
```

Valid Pointer Operations

- Assignment to a pointer of the same type
- Assigning a pointer to pointer of type (void *) and back
- Adding or subtracting a pointer and an integer (including increment and decrement)
- Subtracting or comparing two pointers which point to members of the same array
- Assigning or comparing to zero

Example#8:

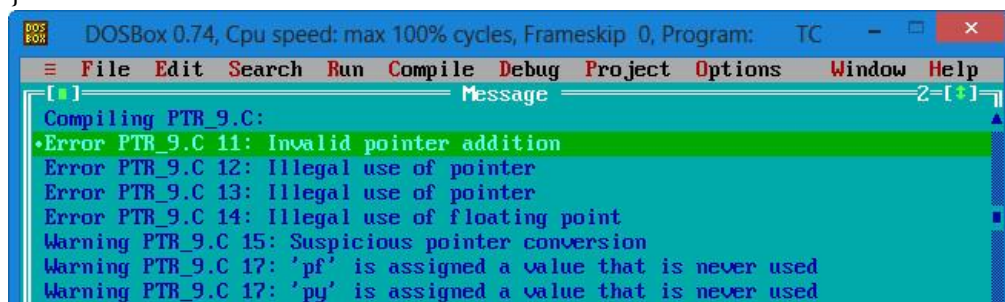
```
#include<stdio.h>
#include<conio.h>
#define NULL 0
void main(){
    int x,y,*px=&x;
    int *py;
    void *pv;
    clrscr();
    py=px;          /* Assignment to a pointer of the same type*/
    px=(int *) pv;  /* recast a (void *) pointer */
    pv=(void *)px;  /* recast to type (void *) */
    py=px+2;        /* Adding or subtracting a pointer and an integer is legal */
    px++;           /* Increment and decrement is legal*/
    if(px==NULL) /* Compare to null pointer*/
    py=NULL;        /* Assign to null pointer*/
}
```

Invalid Pointer Operations

- Adding two pointers
- Multiply, divide, shift, mask pointers
- Add float or double numbers to a pointer
- Assign a pointer of different types without cast

Example#9:

```
main(){
    int x,y,*px,*py,*p;
    float *pf;
    px=&x;
    py=&y;
    p=px+py; /*Addition of two pointer is illegal */
    p=px*py; /* Multiplication of two pointer is illegal */
    p=px/py; /* Division of two pointer is illegal */
    p=px+10.5; /* Addition of float to pointer is illegal */
    pf=px; /* Assignment of different types of pointer is illegal */
}
```



Pointers and Functions

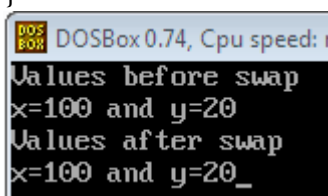
Pointers are very much used in function declaration. Sometimes only with a pointer a complex function can be easily represented and accessed. The use of the pointers in a function definition may be classified into two groups they are call by value and call by reference.

Call By Value

Whenever a portion of the program invokes a function with formal arguments, control will be transferred from the main to the called function and the value of the actual argument is copied to the function. Within the function, the actual value is copied from the calling portion of the program may be altered or changed. When the control is transferred back from the function to the calling portion of the program, the altered values are not transferred back from the function to the calling portion of the program, the altered values are not transferred back. This type of passing formal arguments to a function is technically known as call by value.

Example#10:

```
#include<stdio.h>
#include<conio.h>
void swap(int a,int b){
    int temp;
    temp=a;
    a=b;
    b=temp;
}
void main(){
    int x=100,y=20;
    clrscr();
    printf("Values before swap\n");
    printf("x=%d and y=%d",x,y);
    swap(x,y);
    printf("\nValues after swap\n");
    printf("x=%d and y=%d",x,y);
    getch();
}
```



Since, the above function is declared call by value, the value is not swapped. As a single variable is transferred to the function, it protects the value of this variable from alteration within the function. On the other hand, it prevents the altered value being transferred back from the function to the calling portion of the program.

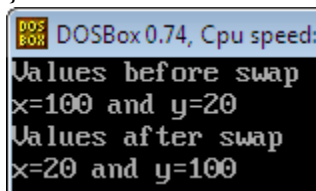
Call by Reference

When a portion of a program calls the function, the addresses of the actual arguments are copied on to the formal arguments, though they may be referred by different variable names. The content of the variables that are altered within the function block are returned to the calling portion of the program in the altered form itself, as the formal and actual arguments are referencing the same memory location or address. The technique is known as call by reference or call by adder or call by location.

When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to the data item within the function is not carried over into the calling routine. When arguments passed by reference(i.e. when a pointer is passed to the function), the address of the data item is passed to the function. The contents of that address can be accessed freely, either within a function or within a calling routine. Moreover, any changes that is made to the data item(i.e. the contents of the address)will be recognized in both the function and the calling portion of the program. Thus, the use of the pointer as function arguments permits the corresponding data item to be altered globally from within the function.

Example#11: A program to exchange the content of two variables using call by reference

```
#include<stdio.h>
#include<conio.h>
void swap(int *a,int *b){
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
void main(){
    int x=100,y=20;
    clrscr();
    printf("Values before swap\n");
    printf("x=%d and y=%d",x,y);
    swap(&x,&y);
    printf("\nValues after swap\n");
    printf("x=%d and y=%d",x,y);
    getch();
}
```



Pointers and Arrays

In C and C++, there is a close correspondence between array datatype and pointers. Any array name in C and C++ is very much like a pointer but there is difference between them. The pointer is variable that can appear on the left side of an assignment operation. The array name is constant and can't appear on the left side of an assignment operator. In all the other aspects, both the pointer and the array are the same.

Recall that an array name is really a pointer to the first element in the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either $\&x[0]$ or simply as x . Moreover the address of the second array element can be written as either $\&x[1]$ or as $(x+1)$, and so on. In general the address of array element $(i+1)$ can be expressed as either $\&x[i]$ or as $(x+i)$. Thus we have two different ways to write the address of an array element: we can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

Since $\&x[i]$ and $(x+i)$ both represent the address of the i th element of x , it would seem reasonable that $x[i]$ and $*(x+i)$ both represent the contents of that address, i.e., the value of the i th element of x . Hence, either term can be used in any particular application.

Table 1: Equivalent Expressions of Arrays & Pointers

Type of Array	To be Accessed	Technique	
		Array Notation	Pointer Notation
One-Dimensional	Address of i^{th} Element	$\&\text{marks}[i]$	$(\text{marks}+i)$
	Value of i^{th} Element	$\text{marks}[i]$	$*(\text{marks}+i)$
Two-Dimensional	Address of Element at i^{th} Row & j^{th} Column	$\&\text{marks}[i][j]$	$*(\text{marks}+i)+j)$
	Value of Element at i^{th} Row & j^{th} Column	$\text{marks}[i][j]$	$*(\text{marks}+i)+j)$

For Example:

```
int x[5]={ 1,2,3,4,5};
int *p;
p=x; /* p=&x[0] */
```

Element	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

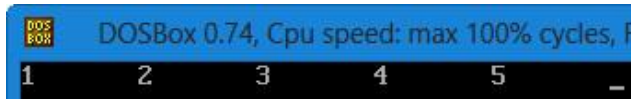
```
p=&x[0];
p=&x[1];
p=&x[2];
p=&x[3];
p=&x[4];
```

Example#12: A program to access the array element using pointer

```

main(){
    int myArray[5]={ 1,2,3,4,5}; int i;
    for(i=0;i<5;i++){
        printf("%d\t",*(myArray+i));
        /* OR */
        /* printf("%d\t",myArray[i]); */
    }
    getch();
}

```



In above program, instead of adding i to myArray to step through the array address, we wanted to use the increment operator. Could we write `*(myArray++)` ? The answer is no, and the reason is that we want increment a constant (or indeed change it in anyway). The expression `myArray` is the address where the system has chosen to place our array, and it will stay at this address until the program terminates. `myArray` is constant. We can't say `myArray++` any more though we can the array element of pointer to which it is pointer can be incremented. While we can't increment an address, we can increment a pointer that holds an address.

```

main(){
    int myArray[]={ 1,2,3,4,5};
    int *ptr2myArray,i; clrscr();
    ptr2myArray=myArray;
    for(i=0;i<5;i++){
        printf("%d ",*(ptr2myArray++));
    }
    getch();
}

```



When assigning a value to an array element such as `x[i]`, the left side of the assignment statement be written as either `x[i]` or as `*(x+i)`. Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element.

On the other hand, it is sometimes necessary to assign an address to an identifier. In such situations, a pointer variable must appear on the left side of the assignment statement. It is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as `x`, `(x+i)` and `&x[i]` cannot appear on the left side of an assignment statement. Moreover, the address of an array cannot arbitrarily be altered, so that expressions such as `++x` are not permitted.

Note that *the address of one array element cannot be assigned to some other array element*. Thus, we cannot write a statement such as

```
&line[2]=&line[1];
```

On the other hand, we can assign the value of one array element to another through a pointer if we wish.

```
p1=&line[1];
line[2]=*p1;
```

OR

```
P1=(line+1);
*(line+2)=*p1;
```

If a numerical array is defined as a pointer variable, the array elements cannot be assigned initial values. Therefore, a conventional array definition is required if initial values will be assigned to the elements of a numerical array. However, a character-type pointer variable can be assigned an entire string as a part of the variable declaration. Thus, a string can conveniently be represented by either a one-dimensional character array or a character pointer.

Example#13:

```
#include<conio.h>
#include<stdio.h>
char x[]="this is first\n\n";
void main(){
    char y[]="this is second\n"; clrscr();
    printf("%s",x); printf("%s",y); getch();
}
```



Example#14:

```
#include<conio.h>
#include<stdio.h>
char *x="this is first\n\n";
void main(){
    char *y="this is second\n"; clrscr();
    printf("%s",x);
    printf("%s",y);
    getch();
}
```



Example#15: Finding smallest element of Array using pointer

```

#include<stdio.h>
#include<conio.h>
void main(){
    int i,n,small,*ptr,a[10]; clrscr();
    printf("How many elements? ");
    scanf("%d",&n);
    for(i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    //Assign address of a[0] to pointer variable
    ptr=a; //It can be done in two ways ptr=&a[0] or ptr=a
    small=*ptr;
    ptr++; //Pointer points to next location in an array
    for(i=1;i<n;i++){ //loop n-1 times to search smallest element
        if(small>*ptr){
            small=*ptr;
        }
        ptr++; //pointer is incremented to pointed a[i+1]
    }
    printf("\nSmallest element of the array: %d",small); getch();
}

```

```

DOS BOX  DOSBox 0.74, Cpu speed: max 100%
How many elements? 5
45 -70 50 1 67
Smallest element of the array: -70_

```

Example#16: Function Returning Pointers

```

#include<stdio.h>
#include<conio.h>
int *larger(int *,int *); /* Prototype */
void main(){
    int a=10,b=20,*p;
    clrscr();
    p=larger(&a,&b); /* Function Call */
    printf("Largest Value=%d",*p);
    getch();
}
int *larger(int *x,int *y){
    if (*x>*y){
        return(x); /* Address of a */
    }else{
        return(y); /* Address of b */
    }
}

```

```

DOS BOX  DOSBox 0.74, Cpu speed:
Largest Value=20_

```

Note:- If we remove pointer operator (*) from the name of user-defined function, then there will be warning: "Nonportable pointer conversion".