

Unit-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.

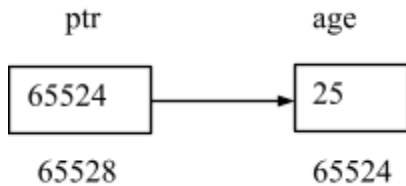
Syntax:

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.



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.

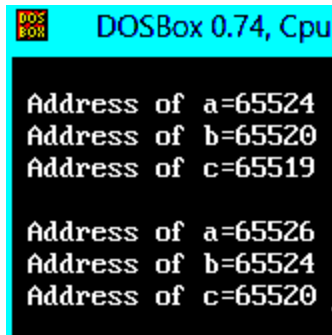
Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative to the size of base type of pointer. For example if we have an integer pointer pi which contains address 1000 then on incrementing we get 1002 instead of 1001. This is because the size of int 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



```
DOSBox 0.74, Cpu
Address of a=65524
Address of b=65520
Address of c=65519

Address of a=65526
Address of b=65524
Address of c=65520
```

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.

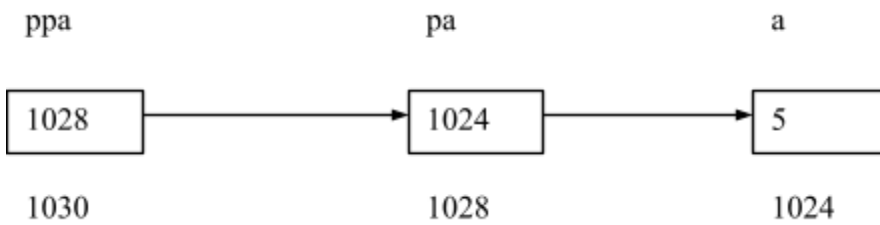
pointer to pointer (double indirection)

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

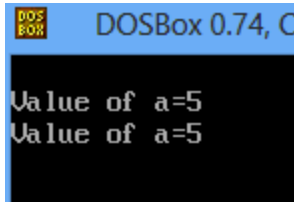
Syntax
 datatype **identifier;
 Eg:
 int **pptr;

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();
}
```



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

```
How many elements are there: 5
Enter element: 45
Enter element: 34
Enter element: 23
Enter element: 12
Enter element: 78
Printing array element:
45
34
23
12
78
```

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

```

How many elements are there?: 5

Enter elements :4

Enter elements :9

Enter elements :18

Enter elements :15

Enter elements :2

Printing sorted elements:
2
4
9
15
18_

```

2. 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

```
How many elements are there? 5

Enter elements of array: 8

Enter elements of array: 4

Enter elements of array: 9

Enter elements of array: 2

Enter elements of array: 6

Sum of all elements of array =29_
```


3. 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

```

How many elements are there? 5

Enter elements :2
Enter elements :4
Enter elements :9
Enter elements :1
Enter elements :3

Enter an element to be searched: 9

9 is found in 3 position_

```

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

```
Enter elements of matrix: 5
Enter elements of matrix: 7
Enter elements of matrix: 9
Enter elements of matrix: 3
Enter elements of matrix: 2
Enter elements of matrix: 8
5      7      9
3      2      8
```

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";
```

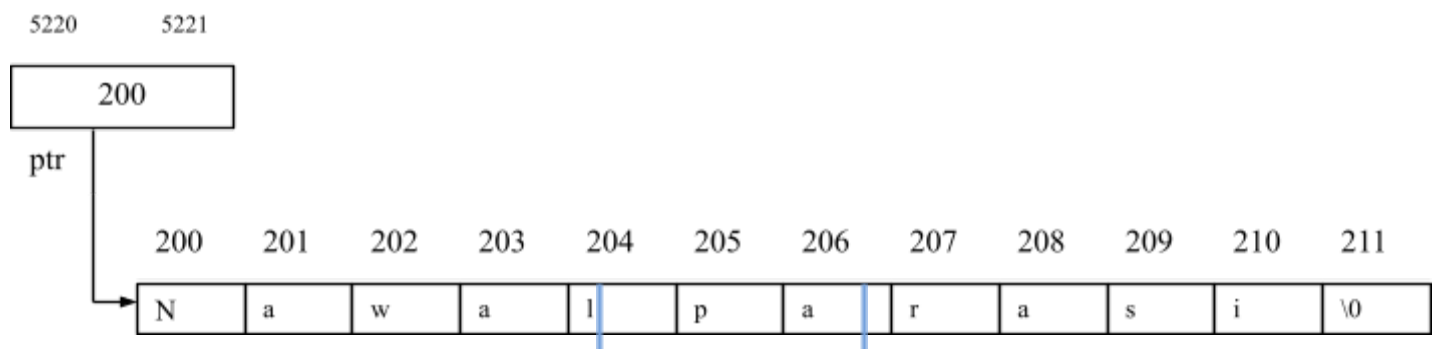
These two forms may look similar but there are some differences in them. The initialization itself has different meaning in both forms. In array form, initialization is a short form for-

```
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.

1000	1001	1002	1003	1004	1005	1006	1007
C	h	i	t	w	a	n	\0
str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]	str[7]



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

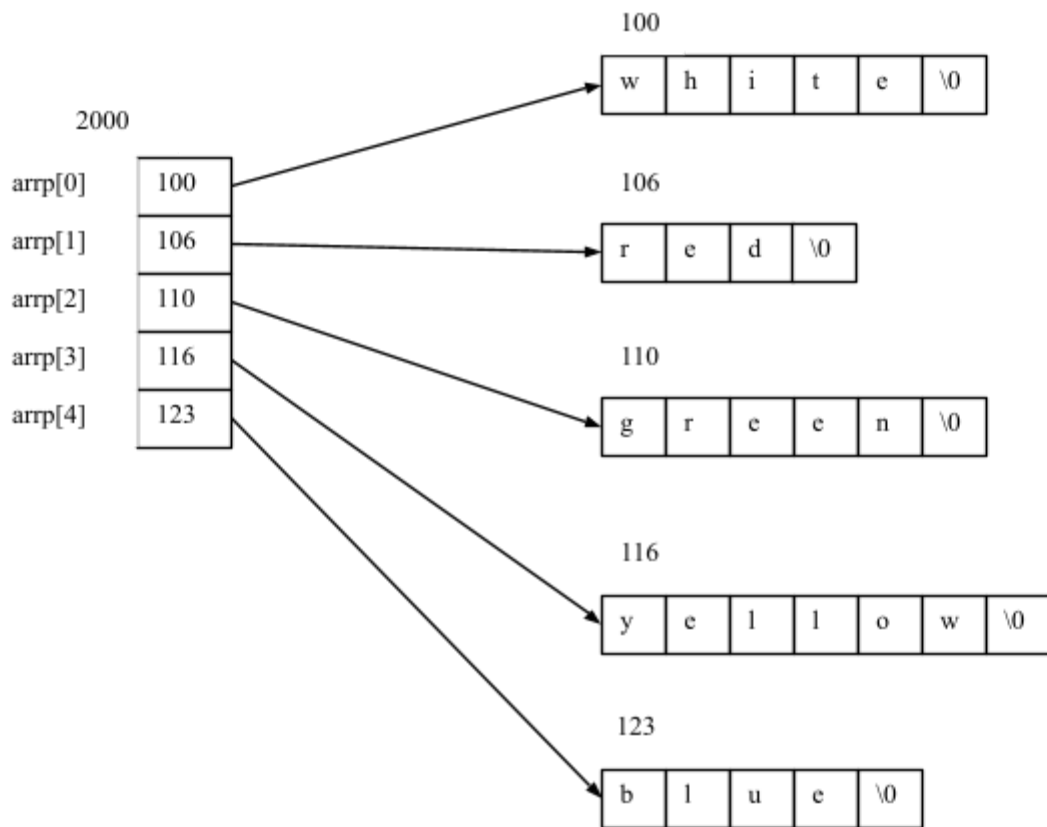
```
str = "Narayangarh"    //invalid
ptr = "Butwal"         //valid
```

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”.



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

Dynamic memory allocation

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.

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.

2. 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.

3. 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.

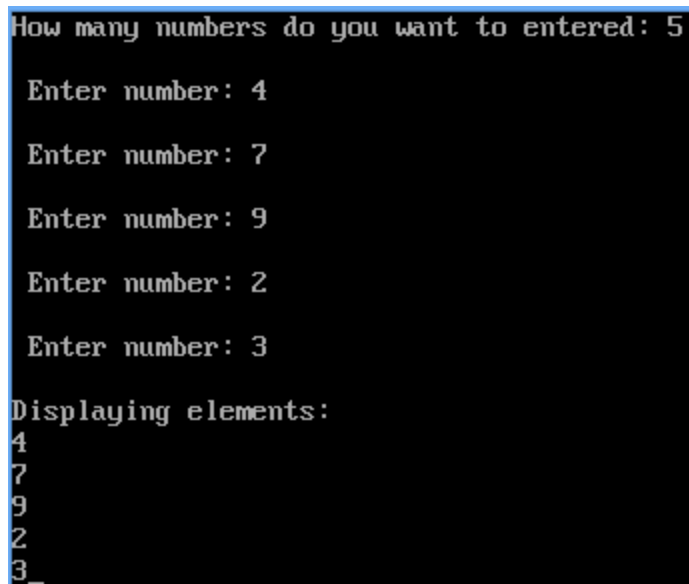
This function takes two arguments, first is a pointer to the block of memory that was previously allocated by `malloc()` or `calloc()` and second one is the new size for that block. For example

```
ptr=(int*) realloc(ptr,newsiz);
```

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



```
How many numbers do you want to entered: 5

Enter number: 4

Enter number: 7

Enter number: 9

Enter number: 2

Enter number: 3

Displaying elements:
4
7
9
2
3_
```

2. 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

How many numbers do you want to entered: 5

Enter number: 2

Enter number: 9

Enter number: 3

Enter number: 8

Enter number: 4

Displaying sorted elements:

2

3

4

8

9